Understanding Deep Learning

Simon J.D. Prince

July 30, 2022

I would really appreciate help improving this document. No detail too small! Please mail suggestions, factual inaccuracies, ambiguities, questions, and errata to udlbookmail@gmail.com.

Contents

Chapter 1

Introduction

Chapter 2

Supervised learning

A *supervised learning model* defines a mapping from one or more inputs to one or more outputs. For example, the input might be the age and mileage of a secondhand Toyota Prius and the output might be the estimated value of the car in dollars.

The model is just a mathematical equation; when the inputs are passed through this equation, it computes the prediction, and this is termed *inference*. This model equation also contains *parameters*. Different parameter values change the outcome of the computation; the model equation describes a family of possible relationships between inputs and outputs, and the parameters specify the particular relationship.

When we *train* or *learn* a model, we choose the parameters so that the relationship between inputs and outputs is correct. The learning algorithm takes a training set of input/output pairs and manipulates the parameters until the inputs predict their corresponding outputs as closely as possible. If the model works well for these training pairs, then we hope that it will make good predictions for new inputs where the true output is unknown.

The goal of this chapter is to expand on these ideas. First, we'll describe this framework more formally and introduce some notation. Then we'll work through a simple example in which we use a straight line to describe the relationship between input and output. This linear model is both familiar and easy to visualize, but nevertheless illustrates all the main ideas of supervised learning.

2.1 Supervised learning overview

In supervised learning, we aim to build a model that takes input data $\mathbf{x}$ and outputs a prediction $\mathbf{y}$. For simplicity, we'll assume that both the input $\mathbf{x}$ and output $\mathbf{y}$ are vectors of a predetermined and fixed size. Moreover, the elements of each vector are always organized in the same way, so in the Prius example above, the input vector $\mathbf{x}$ would always contain the age of the car and then the mileage, in that order. This is known as *structured* or *tabular* data.

Notation
Appendix A

To make the prediction, we need a model $\mathbf{f}[\bullet]$ that takes input $\mathbf{x}$ and returns $\mathbf{y}$:

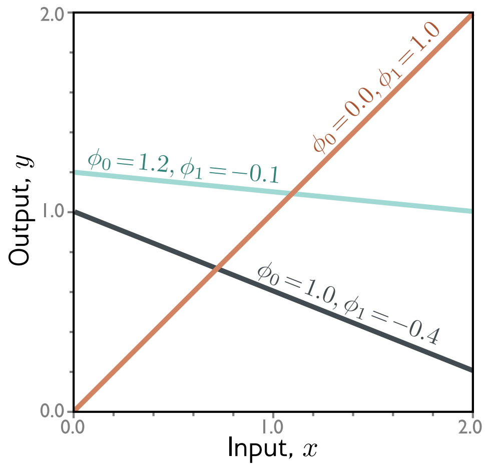


Figure 2.1 Linear regression model. For a given choice of parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model makes a prediction for the output (y-axis) based on the input (x-axis). Different choices for the y-intercept ϕ_0 and the slope ϕ_1 change these predictions (cyan, orange, and gray lines). The linear regression model (equation 2.4) defines a family of input/output relations (lines) and the parameters determine which member of the family (the particular line).

$$\mathbf{y} = \mathbf{f}[\mathbf{x}]. \tag{2.1}$$

When we compute the prediction $\mathbf{y}$ from the input $\mathbf{x}$, we call this *inference*.

The model is just a mathematical equation with a fixed form. It represents a family of different relations between the input and the output. The model also contains *parameters* $\boldsymbol{\phi}$. The choice of parameters determines the particular relation between input and output, and so really, we should write:

$$\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]. \tag{2.2}$$

When we talk about *learning* or *training* a model, we mean that we attempt to find parameters $\boldsymbol{\phi}$ that make sensible output predictions from the input. We learn these parameters using a *training dataset* of I pairs of input and output examples $\{\mathbf{x}_i, \mathbf{y}_i\}$. We aim to select parameters that map each training input to its associated output as closely as possible. We quantify the degree of mismatch in this mapping with the *loss* $\mathrm{L}[\boldsymbol{\phi}]$. This is a scalar value that summarizes how poorly the model predicts the training outputs from their corresponding inputs for parameters $\boldsymbol{\phi}$. So when we train the model, we are seeking parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\mathrm{L}\left[\boldsymbol{\phi}\right]\right]. \tag{2.3}$$

If the final loss is small, then we have found model parameters that accurately predict the training outputs $\mathbf{y}_i$ from the training inputs $\mathbf{x}_i$.

After training the model, we must now assess its performance; we run the model on separate *test data* to see how well it *generalizes* to examples that it did not have access to during training. If the test performance is adequate, then we are ready to apply the model to new data.

2.2 Linear regression example

We'll now make these ideas concrete with a simple example. We'll consider a model $y = \mathrm{f}[x, \boldsymbol{\phi}]$ that predicts a single output y from a single input x. Then we'll develop a loss function, and finally, we'll discuss model training.

2.2.1 1D Linear regression model

A *1D linear regression model* describes the relationship between input x and output y as a straight line:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{2.4}$$

This model has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where ϕ_0 is the y-intercept of the line and the ϕ_1 is the slope. Different choices for the intercept and slope result in different relations between input and output (figure 2.1). Hence, equation 2.4 defines a family of possible input-output relations (all possible lines), and the choice of parameters determines the member of this family (the particular line).

2.2.2 Loss

For this model, the training dataset (figure 2.2a) consists of I input/output pairs $\{x_i, y_i\}$. Figures 2.2b-d show three lines defined by three sets of parameters. The green line in figure 2.2d describes the data more accurately than the other two since it is much closer to the data points. However, we need a principled approach for deciding which parameters $\boldsymbol{\phi}$ are better than others. The goal will be to associate a numerical value with each choice of parameters which we refer to as the *loss*. This quantifies the degree of mismatch between the model and the data; a lower loss means a better fit.

The mismatch is captured by the deviation between the model predictions $\mathrm{f}[x_i, \boldsymbol{\phi}]$ (height of the line at x_i) and the ground truth output values y_i. These deviations are depicted as orange dashed lines in figures 2.2b-d. We quantify the total mismatch, *training error*, or *loss* as the sum of the squares of these deviations for all I training pairs:

$$\begin{aligned} L[\phi] &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2. \end{aligned} \tag{2.5}$$

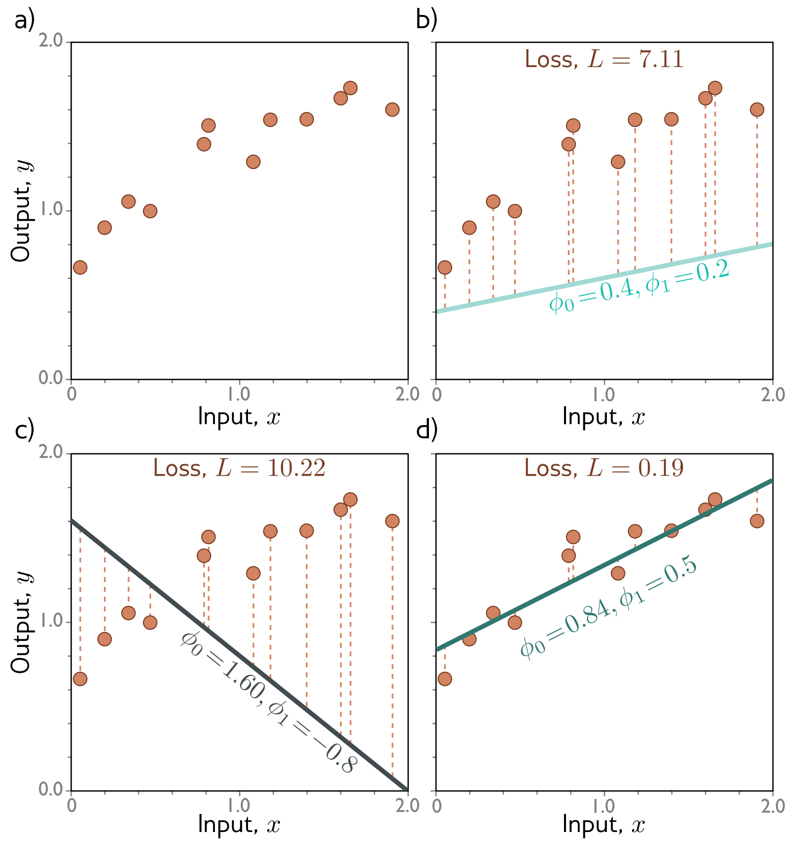


Figure 2.2 Linear regression training data, model, and loss. a) The training data (orange points) consist of $I = 12$ input/output pairs $\{x_i, y_i\}$. b-d) Each panel shows the linear regression model with different parameters. Depending on the choice of slope and intercept parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model errors (orange dashed lines) may be larger or smaller. The loss function $L[\boldsymbol{\phi}]$ is the sum of the squares of these errors. The parameters that define the lines in panels (b) and (c) have large losses $L = 7.11$ and $L = 10.22$ respectively because the model errors are large. The loss $L = 0.19$ in panel (d) is much smaller because the model fits well; in fact, this has the smallest loss of all possible lines and so these are the best possible parameters.

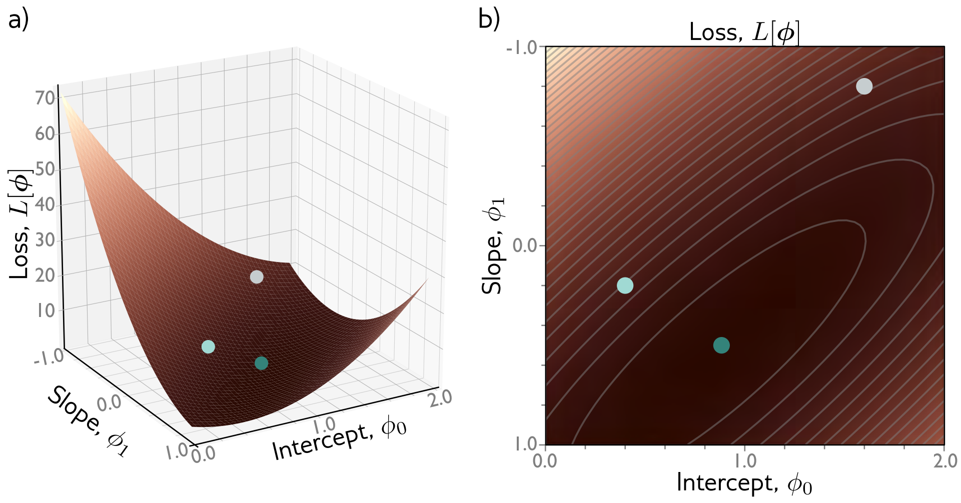


Figure 2.3 Loss function for linear regression model with the dataset in figure 2.2a. a) Each combination of parameters $\boldsymbol{\phi} = [\phi_0,\phi_1]$ has an associated loss. The resulting loss function $L[\boldsymbol{\phi}]$ can be visualized as a surface. The three circles represent the three lines from figure 2.2b-d. b) The loss can also be visualized as a heatmap, where brighter regions represent larger losses; here we are looking straight down at the surface in (a) from above. The best fitting line (figure 2.2d) has the parameters with the smallest loss (green circle).

Since the best parameters minimize this expression, we call this a *least-squares* loss. The squaring operation means that the direction of the deviation (*i.e.*, whether the line is above or below the data) is unimportant. There are also theoretical reasons for this choice which we return to in chapter 5.

The loss L is a function of the parameters $\boldsymbol{\phi}$; it will be larger when the model fit is poor (figure 2.2b,c) and smaller when it is good (figure 2.2d). Considered in this light, we term $L[\boldsymbol{\phi}]$ the *loss function* or *cost function*. The goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this quantity:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[L[\boldsymbol{\phi}]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\left(\mathrm{f}[x_i,\boldsymbol{\phi}]-y_i\right)^2\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\left(\phi_0+\phi_1 x_i-y_i\right)^2\right]. \end{aligned} \tag{2.6}$$

There are only two parameters (the intercept ϕ_0 and slope ϕ_1) and so we can

Problems 2.1-2.2

calculate the loss for every combination of values and visualize the loss function as a surface (figure 2.3). The "best" parameters are at the minimum of this surface.

2.2.3 Training

The process of finding parameters that minimize the loss is termed *model fitting*, *training*, or *learning*. The basic method is to choose the initial parameters randomly and then improve them by "walking down" the loss function until we reach the bottom (figure 2.4). One way to do this is to measure the gradient of the surface at the current position and take a step in the direction that is most steeply downhill. Then we repeat this process until the gradient is flat and we can improve no further.[1]

2.2.4 Testing

Having trained the model, we want to know how well it will perform in the real world. We do this by computing the loss on a separate set of *test data*. The degree to which the prediction accuracy *generalizes* to the test data will depend in part on how representative and complete the training dataset was. However, it will also depend on the model complexity. A simple model might not be able to capture the true relationship between input and output. Conversely, a complex model may describe statistical peculiarities of the training data that are atypical and lead to unusual predictions. This is known as *overfitting*.

2.3 Summary

A supervised learning model is a function $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that relates inputs $\mathbf{x}$ to outputs $\mathbf{y}$. The particular relationship is determined by parameters $\boldsymbol{\phi}$. To train the model, we define a loss function $L[\boldsymbol{\phi}]$ over a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$. This quantifies the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and observed outputs $\mathbf{y}_i$ as a function of the parameters $\boldsymbol{\phi}$. Then we search for the parameters that minimize the loss. We evaluate the model on a different set of test data to see how well it generalizes to new inputs.

In chapters 3-5, we will expand on these ideas. First, we'll tackle the model itself; linear regression has the obvious drawback that it can only describe the relation between input and output as a straight line. To resolve this problem, we introduce shallow neural networks in chapter 3. These are only slightly more

[1]In fact, this iterative approach is not necessary for the linear regression model. Here, it's possible to find closed form expressions for the parameters. However, this *gradient descent* approach works in more complex models, where there is no closed from solution, and where there are too many parameters to evaluate the loss for every combination of values.

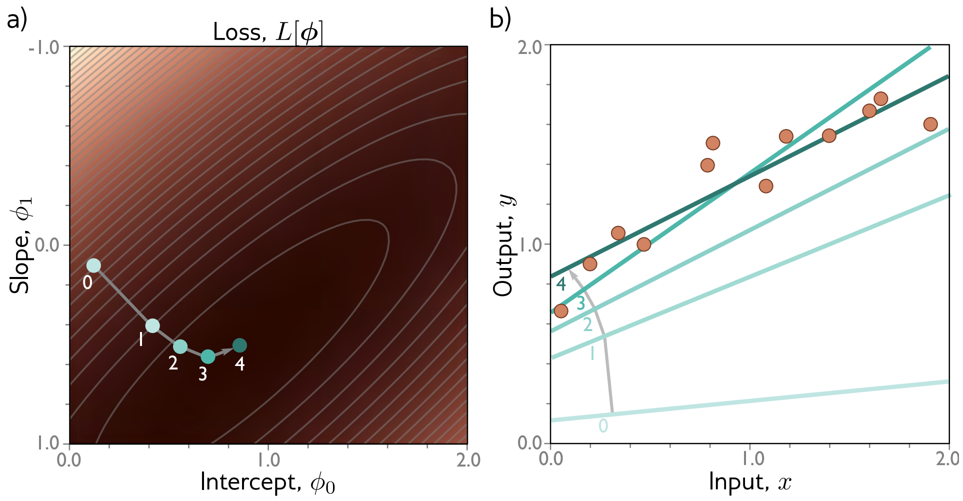


Figure 2.4 Training the linear regression model. The goal is to find the slope/intercept parameters that correspond to the smallest loss. a) Iterative training algorithms initialize the parameters randomly and then improve them by "walking downhill" until no further improvement can be made. Here we start at position 0 and move a certain distance downhill (perpendicular to the contours) to position 1. Then we re-calculate the downhill direction and move to position 2. Eventually, we wind up at the minimum of the function (position 4). b) Each position 0-4 from panel (a) corresponds to a different y-intercept and slope and hence represents a different line. As the loss decreases, these lines fit the data more closely.

complex than linear regression but describe a much larger family of relationships between the input and output. In chapter 4 we'll introduce deep neural networks, which are just as expressive, but can describe complex functions with fewer parameters and work better in practice.

In chapter 5 we'll investigate loss functions for different tasks and reveal the theoretical underpinnings of the least-squares loss function. In chapters 6 and 7, we'll discuss the training process and introduce the famous backpropagation algorithm. In chapter 8 we'll discuss how to measure model performance. In chapter 9, we'll consider *regularization* techniques, which aim to improve that performance.

Notes

Loss functions vs. cost functions: In much of machine learning and in this book, the terms loss function and cost function are used interchangeably. However, more properly a loss function is the individual term associated with a data point (*i.e.*, each of the squared terms on the right-hand side of equation 2.5) and the cost function is the overall quantity that is minimized (*i.e.*, the entire right-hand side of equation 2.5). A cost function can contain additional terms that are not associated with individual data points. More generally, an *objective function* is any function that is to be maximized or minimized.

Generative vs. discriminative models: The models $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ in this chapter are *discriminative models*. These make an output prediction $\mathbf{y}$ from real-world measurements $\mathbf{x}$. Another approach is to build a *generative model* $\mathbf{x} = \mathbf{g}[\mathbf{y}, \boldsymbol{\phi}]$, in which the real-world measurements $\mathbf{x}$ are computed as a function of the output $\mathbf{y}$.

Problem 2.3

The generative approach has the disadvantage that it doesn't directly predict $\mathbf{y}$. To perform inference, we must invert the generative equation as $\mathbf{y} = \mathbf{g}^{-1}[\mathbf{x}, \boldsymbol{\phi}]$ and this may be difficult. However, generative models have the advantage that we can build in prior knowledge about how the data were generated. For example, if we wanted to predict the 3D position and orientation $\mathbf{y}$ of a car in an image $\mathbf{x}$, then we could build knowledge about car shape, 3D geometry, and light transport into the function $\mathbf{x} = \mathbf{g}[\mathbf{y}, \boldsymbol{\phi}]$.

This seems like a good idea, but in fact, discriminative models dominate modern machine learning; any advantage gained from exploiting prior knowledge in generative models is usually trumped by brute force learning of very flexible discriminative models from large amounts of training data.

Problems

Problem 2.1 To walk 'downhill' on the loss function (equation 2.5), we measure its slope with respect to the parameters ϕ_0 and ϕ_1. Calculate expressions for the slopes $\partial L/\partial\phi_0$ and $\partial L/\partial\phi_1$.

Problem 2.2 Show that we can find the minimum of the loss function in closed form by setting the expression for the derivatives from problem 2.1 to zero and solving for ϕ_0 and ϕ_1. Note that this works for linear regression but not for more complex models; this is why we use iterative model fitting methods like gradient descent (figure 2.4).

Problem 2.3 Consider reformulating linear regression as a generative model so we have $x = \mathrm{g}[y, \boldsymbol{\phi}] = \phi_0 + \phi_1 y$. What is the new loss function? Find an expression for the inverse function $y = \mathrm{g}^{-1}[x, \boldsymbol{\phi}]$ that we would use to perform inference. Will this model make the same predictions as the discriminative version for a given training dataset $\{x_i, y_i\}$?

Chapter 3

Shallow neural networks

Chapter 2 introduced supervised learning using 1D linear regression. However, this model can only describe the input/output relationship as a line. This chapter introduces shallow neural networks. These describe piecewise linear functions and are flexible enough to describe arbitrarily complex relationships between multi-dimensional inputs and outputs.

3.1 Neural network example

In their general form, shallow neural networks are functions $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that map multivariate inputs $\mathbf{x}$ to multivariate outputs $\mathbf{y}$. We'll defer a full definition until section 3.4 and introduce the main ideas using a simple example in which we map a scalar input x to a scalar output y so $y = \mathrm{f}[x, \boldsymbol{\phi}]$:

$$y = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]. \tag{3.1}$$

This model has ten parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$.

The calculation in equation 3.1 can be broken down into three parts: first, we compute three linear functions of the input data ($\theta_{10} + \theta_{11}x$, $\theta_{20} + \theta_{21}x$, and $\theta_{30} + \theta_{31}x$). Second, we pass each of the three results through an *activation function* $\mathrm{a}[\bullet]$. Finally, we weight the three resulting activations with ϕ_1, ϕ_2, and ϕ_3 respectively, sum them, and add an offset ϕ_0.

To complete the description, we must define the activation function $\mathrm{a}[\bullet]$. There are several possibilities, but the most common is the *rectified linear unit* or *ReLU*: Problem 3.1

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}. \tag{3.2}$$

This returns the input when it is positive and zero otherwise (figure 3.1).

It's probably not obvious which family of input/output relations is represented by equation 3.1. Nonetheless, the ideas from the previous chapter are all applicable.

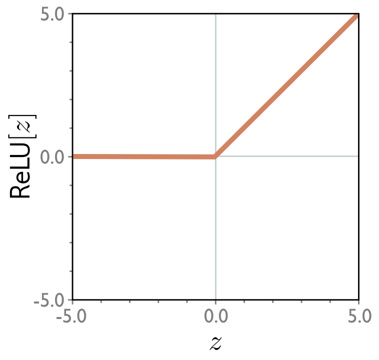


Figure 3.1 Rectified linear unit (ReLU). This activation function returns zero if the input is less than zero and returns the input unchanged if it is greater than zero. In other words, it clips negative values to zero. Note that there are many other possible choices for the activation function (see figure 3.13) but the ReLU is the most used and easiest to understand.

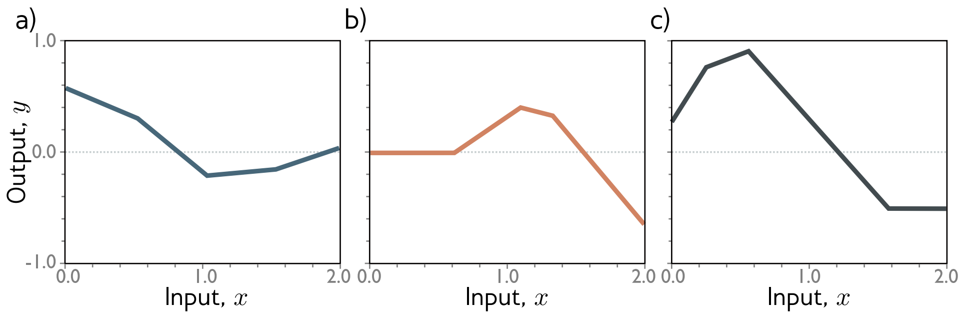


Figure 3.2 Family of functions defined by equation 3.1. a-c) Functions for three different choices of the ten parameters $\boldsymbol{\phi}$. In each case, the input/output relation is piecewise linear. However, the positions of the joints, the slopes of the linear regions between them, and the overall height vary.

Equation 3.1 represents a family of functions, where the particular member of the family depends on the ten parameters in $\boldsymbol{\phi}$. If we know these parameters, we can perform inference (predict y) by evaluating the equation for a given input x. Given a training dataset $\{x_i, y_i\}_{i=1}^{I}$, we can define a least squares loss function $\mathrm{L}[\boldsymbol{\phi}]$, and so measure how effectively the model describes this dataset for any given parameter values $\boldsymbol{\phi}$. To train the model, we search for the values $\hat{\boldsymbol{\phi}}$ that minimize this loss.

3.1.1 Neural network intuition

In fact, equation 3.1, represents a family of continuous piecewise linear functions (figure 3.2) with up to four linear regions. We'll now break down equation 3.1 and show *why* it describes this family. To make this easier to understand, we'll split the function into two parts. First, we introduce the intermediate quantities:

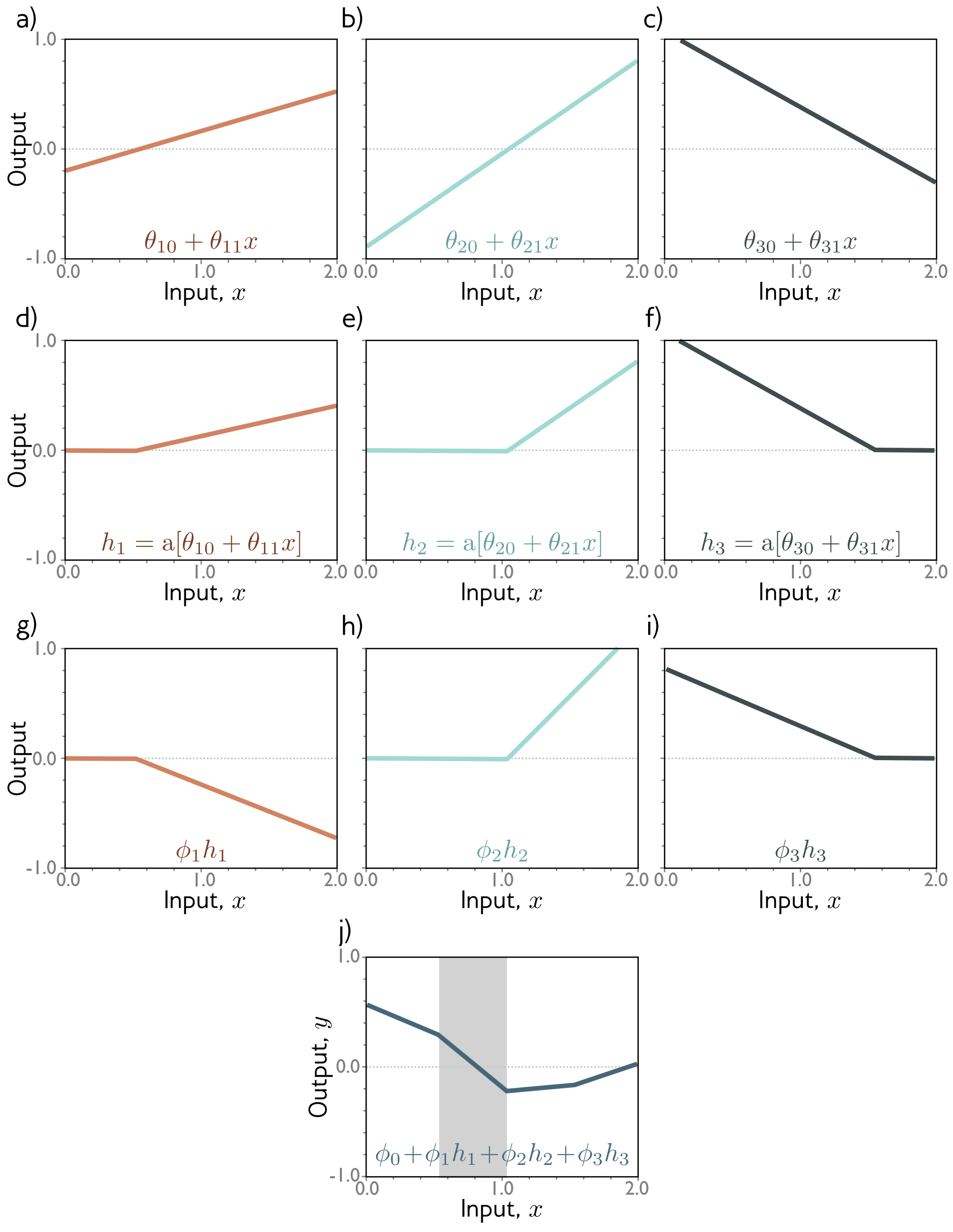


Figure 3.3 Computation for function in figure 3.2a. a-c) The input x is passed through three linear functions, each with a different y-intercept $\theta_{\bullet 0}$ and slope $\theta_{\bullet 1}$. d-f) Each line is passed through the ReLU activation function, which clips negative values to zero. g-i) The three clipped lines are then weighted (scaled) by the parameters ϕ_1, ϕ_2, and ϕ_3 respectively. h) Finally, the clipped and weighted functions are summed together and an offset ϕ_0 that controls the overall height is added. Each of the four linear regions corresponds to a different activation pattern in the hidden units. In the shaded region, h_2 is inactive (clipped) but h_1 and h_3 are both active.

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{3.3}$$

where we refer to h_1, h_2, and h_3 as *hidden units*. Second, we compute the output by combining these hidden units with a linear function:[1]

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{3.4}$$

Figure 3.3 shows the flow of computation that creates the function in figure 3.2a. Each hidden unit contains a linear function $\theta_{\bullet 0} + \theta_{\bullet 0}x$ of the input, and that line is clipped by the ReLU function $a[\bullet]$ below zero. The positions where the three lines cross zero become the three "joints" in the final output. The three clipped lines are then weighted by ϕ_1, ϕ_2, and ϕ_3 respectively. Finally, the offset ϕ_0 is added, which controls the overall height of the final function.

Problems 3.2-3.8

Each linear region in figure 3.3j corresponds to a different *activation pattern* in the hidden units. When a unit is clipped, we refer to it as *inactive*, and when it is not clipped, we refer to it as *active*. For example, the shaded region receives contributions from h_1 and h_3 (which are active) but not from h_2 (which is inactive). The slope of each linear region is determined by (i) the original slopes $\theta_{\bullet 1}$ of the active inputs for this region, and (ii) the weights $\phi_{\bullet}$ that were subsequently applied. For example, the slope in the shaded region is $\theta_{11}\phi_1 + \theta_{31}\phi_3$.

Each hidden unit contributes one 'joint' to the function, so with three hidden units, there can be four linear regions. However, only three of the slopes of these regions are independent; the fourth is either zero (if all the hidden units are inactive in this region) or is a sum of slopes from the other regions.

Problem 3.9

3.1.2 Depicting neural networks

We have been discussing a neural network with one input, one output, and three hidden units. We visualize this network in figure 3.4a. The input is on the left, the hidden units are in the middle, and the output is on the right. Viewed in this way, each connection represents one of the ten parameters. To simplify this representation, we do not typically draw the intercept parameters, and so this network would usually be depicted as in figure 3.4b.

[1]A linear function has the form $z' = \phi_0 + \sum_i \phi_i z_i$. Any other type of function is non-linear. For instance, the ReLU function (equation 3.2) and the example neural network that contains it (equation 3.1) are both non-linear.

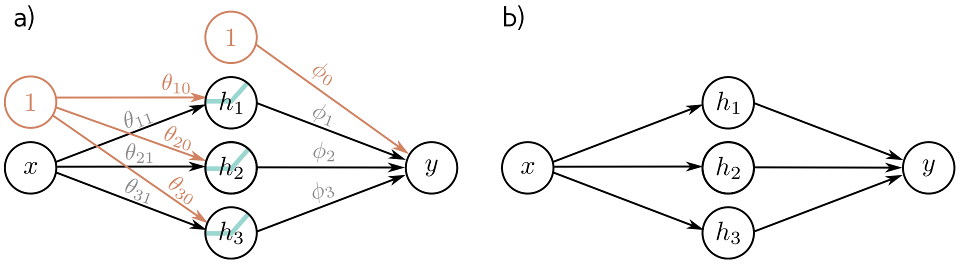


Figure 3.4 Depicting neural networks. a) The input x is on the left and the output y on the right. The three hidden units are in the center. The computation flows from left to right. The input is used to compute the values of the three hidden units and these values are combined to create the output. Each of the ten arrows represents one of the parameters. The intercepts are in orange and the slopes in black. Each parameter multiplies its source variable and adds the result to its target variable. For example, we multiply the parameter ϕ_1 by its source h_1 and add it to y. We introduce additional nodes containing ones (orange circles) to incorporate the offsets. The ReLU functions are applied in the hidden units. b) More typically, the intercepts, ReLU functions, and parameter names are omitted, and so this simpler depiction would be used to represent the same network.

3.2 Universal approximation theorem

In the previous section, we introduced an example neural network with one input, one output, ReLU activation functions, and three hidden units. Let's not now generalize this slightly and consider the more general case with D hidden units where the d^{th} hidden unit is:

$$h_d = \mathrm{a}[\theta_{d0} + \theta_{d1}x], \tag{3.5}$$

and these are combined linearly to create the output:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d. \tag{3.6}$$

The number of hidden units in a shallow network is a measure of the *network capacity*. With ReLU activation functions, the output of a network with D hidden units is a piecewise linear function with at most $D+1$ linear regions. As we increase the number of hidden units, the model can approximate more complex functions. Problem 3.10

Indeed, with enough capacity (hidden units), a shallow network can describe any continuous 1D function defined on a compact subset of the real line to arbitrary precision. To see this, consider that every time we add a hidden unit, we add another linear region to the function. As these linear regions become more numerous, they represent smaller and smaller sections of the function, which are

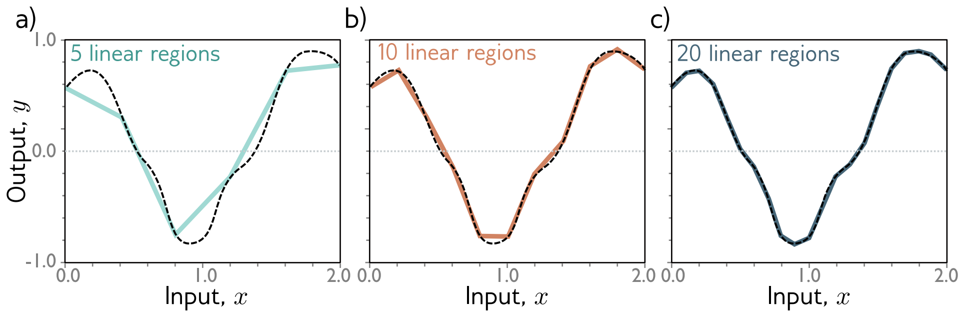


Figure 3.5 Approximation of 1D function (dashed line) by piecewise linear model. a-c) As the number of regions increases, the model becomes closer and closer to the continuous function. A neural network with a scalar input creates one extra piecewise linear region per hidden unit. The universal approximation theorem provides a formal proof that with enough hidden units a shallow neural network can describe any continuous function defined on a compact subset of $\mathbb{R}^D$ to arbitrary precision.

increasingly well approximated by a line (figure 3.5). The ability of a neural network to approximate any continuous function can be formally proven and this is known as the *universal approximation theorem*.

3.3 Multivariate inputs and outputs

In the above example, the network has a single scalar input x and a single scalar output y. However, the universal approximation theorem also holds for the more general case where the network maps multivariate inputs $\mathbf{x} = [x_1, x_2, \ldots x_{D_i}]^T$ to multivariate output predictions $\mathbf{y} = [y_1, y_2, \ldots y_{D_o}]^T$. In this section, we explore how the model can be extended to predict multivariate outputs. Then we'll consider multivariate inputs. Finally, in section 3.4 we'll present a general definition of a shallow neural network.

3.3.1 Visualizing multivariate outputs

To extend the network to multivariate outputs $\mathbf{y}$, we simply use a different linear function of the hidden units for each output. So, a network with a scalar input x, four hidden units h_1, h_2, h_3, and h_4, and a 2D multivariate output $\mathbf{y} = [y_1, y_2]^T$ would be defined as:

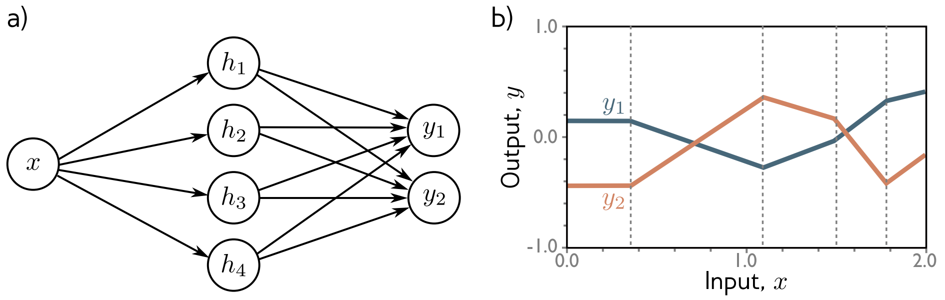


Figure 3.6 Network with one input, four hidden units, and two outputs. a) Visualization of network structure. b) This network produces two piecewise linear functions $y_1[x]$ and $y_2[x]$. The four 'joints' of these functions (at vertical dotted lines) are constrained to be in the same place since they share the same hidden units, but the slopes and overall height may differ.

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \\ h_4 &= \mathrm{a}[\theta_{40} + \theta_{41}x], \end{aligned} \tag{3.7}$$

and

$$\begin{aligned} y_1 &= \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4 \\ y_2 &= \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4. \end{aligned} \tag{3.8}$$

The two outputs are two different linear functions of the hidden units.

As we saw in figure 3.3, the 'joints' in the piecewise functions are determined by where the initial linear functions $\theta_{\bullet 0} + \theta_{\bullet 1}x$ are clipped by the ReLU functions $\mathrm{a}[\bullet]$ at the hidden units. Since both outputs $y_1[x]$ and $y_2[x]$ are different linear functions of the same four hidden units, it follows that the four 'joints' in each must be in the same place. However, the slopes of the linear regions and the overall vertical offset will differ in general (figure 3.6).

Problem 3.11

3.3.2 Visualizing multivariate inputs

To cope with multivariate inputs $\mathbf{x}$, we extend the linear relations between the input and the hidden units. So a network with two inputs $\mathbf{x} = [x_1, x_2]^T$ and a scalar output y (figure 3.7) might have three hidden units that are defined by:

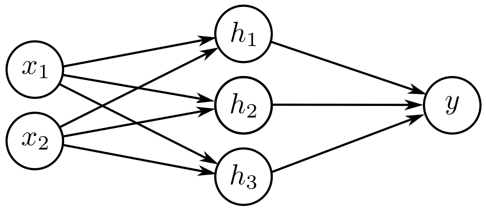


Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2], \end{aligned} \tag{3.9}$$

where there is now one slope parameter for each input. The hidden units are combined to form the output in the usual way:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{3.10}$$

Problems 3.12-3.13

Figure 3.8 illustrates the processing of this network. Each hidden unit receives a linear combination of the two inputs, which forms an oriented plane in the 3D input/output space. The activation function clips negative values of these planes to zero. These clipped planes are then recombined in a second linear transformation (equation 3.10) to create a continuous piecewise linear surface consisting of convex polygonal regions (figure 3.8j). Each region corresponds to a different activation pattern. For example, in the central triangular region, the first and third hidden units are active and the second is inactive.

When there are more than two inputs to the model, it becomes difficult to visualize. However, the interpretation is similar. The output will be a continuous piecewise linear function of the input, where the linear regions are now convex polytopes in the multi-dimensional input space.

Note that as the input dimensions grow, the number of linear regions increases rapidly (figure 3.9). To get a feeling for just how rapidly, consider that each hidden unit defines a hyperplane that delineates the part of space where this unit is active from the part where it is not (cyan lines in 3.8d-f). If we had the same number of hidden units as input dimensions D_i, then we could align each hyperplane with one of the coordinate axes (figure 3.10). For two input dimensions, this would divide the space into four quadrants, for three dimensions this would create eight octants, and for D_i dimensions this would create 2^{D_i} orthants. Shallow neural networks usually have a larger number of hidden units than input dimensions, so they typically create more than 2^{D_i} linear regions.

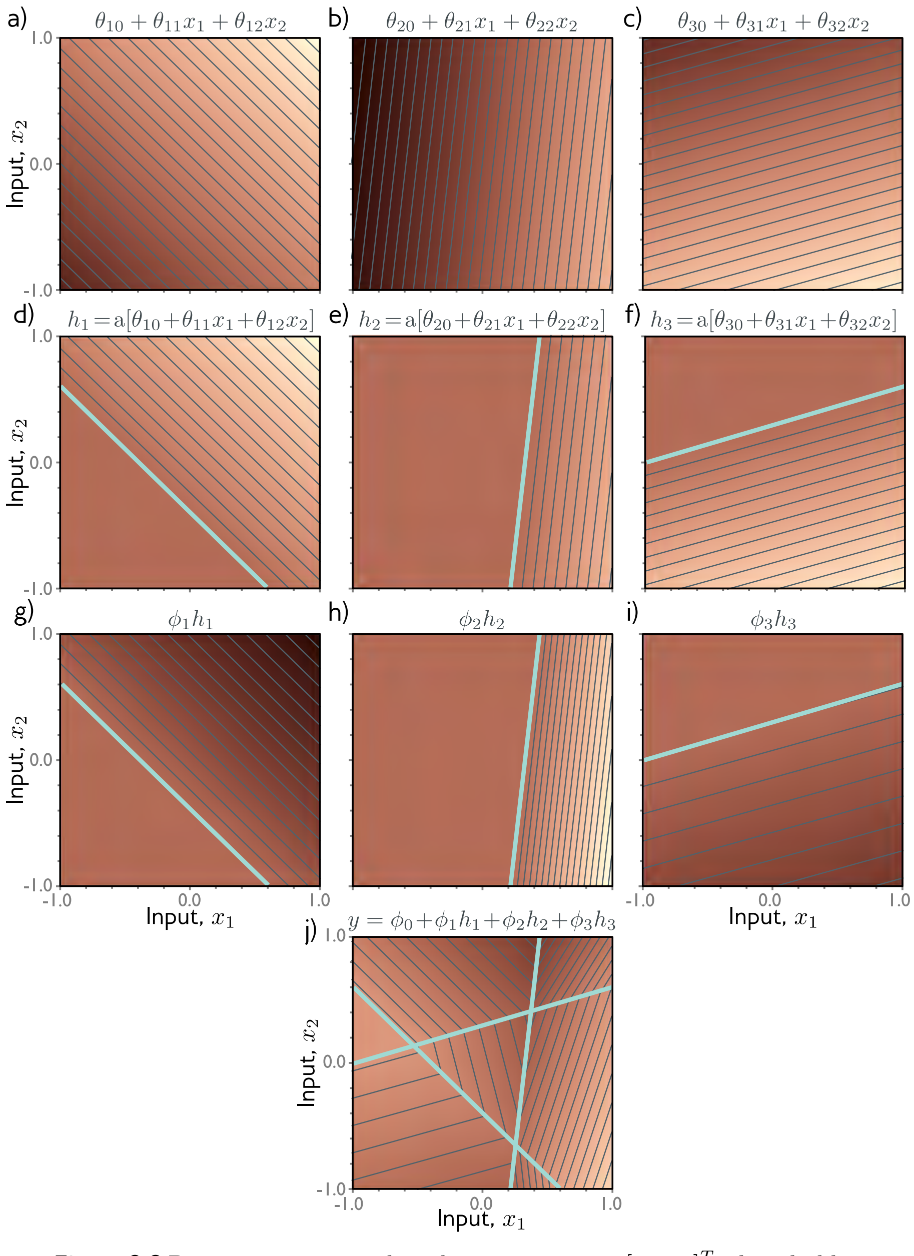


Figure 3.8 Processing in network with two inputs $\mathbf{x}=[x_1,x_2]^T$, three hidden units h_1,h_2,h_3, and one output y. a-c) The input to each hidden unit is a linear function of the two inputs, which corresponds to an oriented plane. d-f) Each plane is clipped by the ReLU activation function (cyan lines are equivalent to 'joints' in figures 3.3d-f). g-h) The clipped planes are then weighted, and j) summed together with an offset that determines the overall height of the surface. The result is a continuous surface made up of convex piecewise linear polygonal regions. Here there are seven linear regions, each of which corresponds to a different activation pattern.

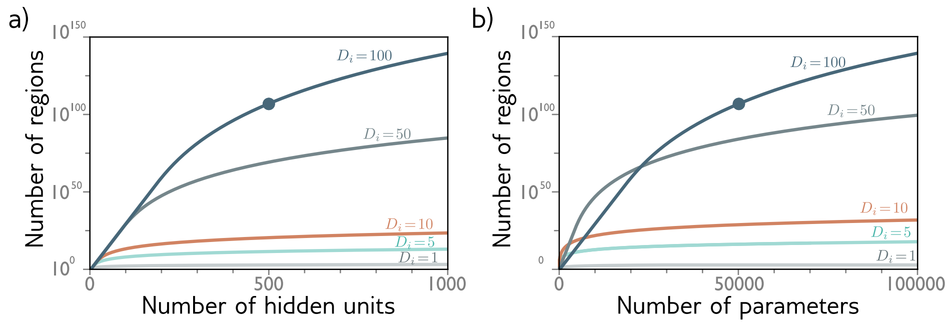


Figure 3.9 Linear regions vs. hidden units. a) Maximum possible regions as a function of the number of hidden units for five different input dimensions $D_i = \{1, 5, 10, 50, 100\}$. The number of regions increases very rapidly in high dimensions; with $D = 500$ units and input size $D_i = 100$, there can be greater than 10^{107} regions (solid circle). b) The same data plotted as a function of the number of parameters. The solid circle represents the same model as in panel (a) with $D = 500$ hidden units. This network has $51,001$ parameters and would be considered very small by modern standards.

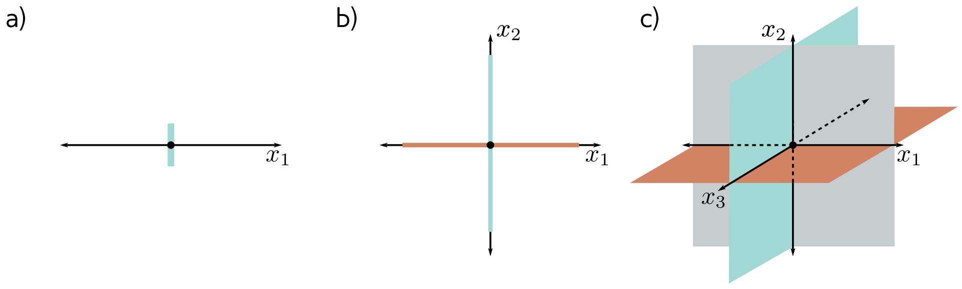


Figure 3.10 Number of linear regions vs. input dimensions. a) With a single input dimension, a model with one hidden unit creates one joint, which divides the axis into two linear regions. b) With two input dimensions, a model with two hidden units can divide the input space using two lines (here aligned with axes) to create four regions. c) With three input dimensions, a model with three hidden units can divide the input space using three planes (again aligned with axes) to create eight regions. Continuing this argument, it follows that a model with D_i input dimensions and D_i hidden units can divide the input space with D_i hyperplanes to create 2^{D_i} linear regions.

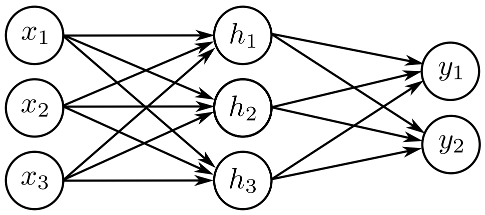


Figure 3.11 Visualization of neural network with three inputs and two outputs. This network has twenty parameters. There are fifteen slopes (indicated by arrows) and five offsets (not shown).

3.4 Shallow neural networks: general case

In this chapter, we have seen several examples of shallow neural networks, which help develop intuition about exactly how they work. We now define a general equation for a shallow neural network $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that maps a multidimensional input $\mathbf{x} \in \mathbb{R}^{D_i}$ to a multidimensional output $\mathbf{x} \in \mathbb{R}^{D_o}$ using $\mathbf{h} \in \mathbb{R}^{D}$ hidden units. Each hidden unit is computed as:

$$h_d = \text{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right], \tag{3.11}$$

and these are combined linearly to create the output:

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d, \tag{3.12}$$

where $a[\bullet]$ is a non-linear activation function. The model has parameters $\boldsymbol{\phi} = \{\theta_{\bullet\bullet}, \phi_{\bullet\bullet}\}$. Figure 3.11 shows a concrete example where there are three inputs, three hidden layers, and two outputs. Problems 3.14-3.17

The purpose of the activation function is to permit the model to describe non-linear relations between the input and the output, and as such it must be non-linear itself; with no activation function, or a linear activation function, the overall mapping from input to output would be restricted to be linear.

Many different activation functions have been tried (figure 3.13), but the most common choice is the ReLU and this has the merit of being easily interpretable. With ReLU activations, the network divides the input space into convex polytopes that are defined by the intersections of hyperplanes computed by the 'joints' in the ReLU functions. Each of these convex polytopes contains a different linear function. The polytopes would be the same for each of the D_o outputs, but the linear functions that they contain would differ.

3.5 Terminology

We conclude this chapter by introducing some terminology. Regrettably, neural networks have a lot of associated jargon (figure 3.12). They are often referred to in terms of *layers*. The left of figure 3.4b is the *input layer*, the center is the *hidden*

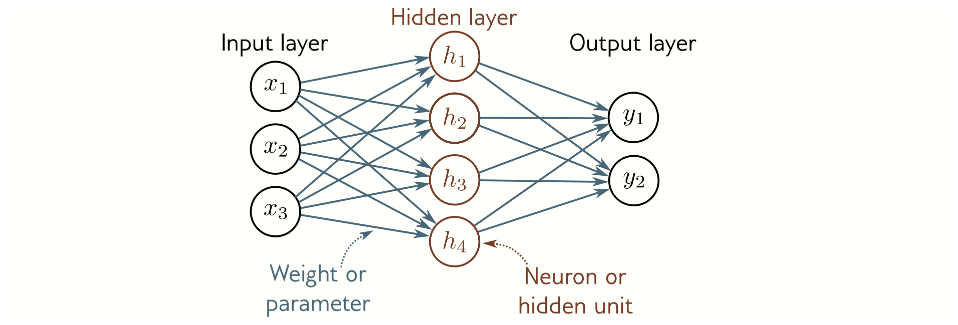


Figure 3.12 Neural network terminology. A shallow network consists of an input layer, a hidden layer, and an output layer. Each layer is connected to the next by forward connections (arrows). For this reason, these models are referred to as feed-forward networks. When every variable in one layer connects to every variable in the next, we refer to this as a fully connected network. Each connection represents a slope parameter in the underlying equation and these parameters are referred to as weights. The variables in the hidden layer are termed neurons or hidden units. The values feeding into the hidden units are termed pre-activations and the values at the hidden units (*i.e.*, after the ReLU function is applied) are termed activations.

layer, and to the right is the *output layer*. We would say that the network in figure 3.4b has one hidden layer containing three hidden units. The hidden units themselves are sometimes referred to as *neurons*. When we pass data through the network, the values of the inputs to the hidden layer (*i.e.*, before the ReLU functions are applied) are termed *pre-activations*. The values at the hidden layer (*i.e.*, after the ReLU functions are applied) are termed *activations*.

For historical reasons, any neural network with at least one hidden layer may also be referred to as a *multi-layer perceptron* or *MLP* for short. Networks with a single hidden layer (as described in this chapter) are sometimes referred to as *shallow neural networks*. Networks with multiple hidden layers (as described in the next chapter) are referred to as *deep neural networks*. Neural networks in which the connections form an acyclic graph (*i.e.*, a graph with no loops, as in all the examples in this chapter) are referred to as *feed-forward networks*. If every element in one layer connects to every element in the next layer (as in all the examples in this chapter), we say that the network is *fully connected.* These connections represent slope parameters in the underlying equations and are referred to as *network weights*. The offset parameters (not shown in figure 3.12) are referred to as *biases*.

3.6 Summary

Shallow neural networks have one hidden layer. They (i) compute several linear transformations of the input, (ii) pass each result through an activation function, and then (iii) take a linear combination of these activations to form the outputs. Shallow neural networks make predictions $\mathbf{y}$ about the world based on inputs $\mathbf{x}$ by dividing the input space into a continuous surface of piecewise linear regions. With enough hidden units (neurons), shallow neural networks can approximate any continuous function to arbitrary precision.

In the next chapter, we'll discuss deep neural networks. These extend the models from this chapter by adding more than one hidden layer. In chapters 5-7, we'll describe techniques to train these models.

Notes

Why "neural" networks? If the models in this chapter are just functions, why are they called "neural networks"? The connection is unfortunately rather tenuous. Visualizations like that in figure 3.12, consist of a set of nodes (inputs, hidden units, and outputs) that are densely connected to one another. This bears a superficial similarity to neurons in the mammalian brain, which also have dense connections. However, the analogy ends there. There is scant evidence that brain computation works in the same way as neural networks, and it is unhelpful to think about biology going forward.

History of neural networks: McCulloch & Pitts (1943) first came up with the notion of an artificial neuron that combined inputs to produce an output, but this model did not have a practical associated learning algorithm. Rosenblatt (1958) developed the *perceptron*, which linearly combined inputs and then thresholded them to make a yes/no decision. He also provided an algorithm to learn the weights of the linear function from data. Minsky & Papert (1969) argued that the linear function was inadequate for general classification problems, but that adding hidden layers with non-linear activation functions (hence the term multi-layer perceptron) could allow the learning of more general input/output relations. However, they concluded that Rosenblatt's algorithm could not learn the parameters of such models. It was not until the 1980s that a practical algorithm (backpropagation, see chapter 7) was developed and significant work on neural networks resumed. The history of the development of neural networks is chronicled by Kurenkov (2020) and Sejnowski (2018).

Activation functions: The ReLU function has been used as far back as Fukushima (1969). However, in the early days of neural networks, it was more common to use the logistic sigmoid or tanh activation functions (figure 3.13a). The ReLU was re-popularized by Jarrett *et al.* (2009), Nair & Hinton (2010), and Glorot *et al.* (2011) and is an important part of the success story of modern neural networks. It has the nice property that the derivative of the output with respect to the input is always one for inputs greater than zero. This contributes to the stability and efficiency of training (see chapter 7) and contrasts with the derivatives of the earlier sigmoid activation functions, which saturate (become close to zero) for large inputs.

However, the ReLU function has the disadvantage that its derivative is zero for negative inputs. If all the training examples produce negative inputs to a given ReLU function, then we cannot improve the parameters feeding into this during training. The gradient with respect to the incoming weights is locally flat and so we cannot "walk downhill". This is known as the *dying ReLU* problem. To resolve this problem, many variations on the ReLU have been proposed (figure 3.13b) including the (i) the leaky ReLU (Maas *et al.* 2013), which also has a linear output for negative values with a smaller slope of 0.1, (ii) the parametric ReLU (He *et al.* 2015), which treats the slope of the negative portion as an unknown parameter and (iii) the concatenated ReLU (Shang *et al.* 2016), which produces two outputs, one of which clips below zero (*i.e.*, like a typical ReLU) and one of which clips above zero.

A variety of continuous functions have also been investigated (figure 3.13c-d), including the softplus function (Glorot *et al.* 2011), Gaussian error linear unit (Hendrycks & Gimpel 2016), sigmoid linear unit (Hendrycks & Gimpel 2016), and exponential linear unit (Clevert *et al.* 2015). Most of these are attempts to avoid the dying neuron problem while limiting the gradient for negative values. Klambauer *et al.* (2017) introduced the scaled exponential linear unit (figure 3.13e), which is particularly interesting as it helps stabilize the variance of the activations when the input variance has a limited range (see section 7.3). Ramachandran *et al.* (2017) adopted an empirical approach to choosing an activation function. They searched the space of possible functions to find the one that performed best over a variety of supervised learning tasks. The optimal function was found to be $a[x] = x/(1+\exp[-\beta x])$ where β is a learned parameter (figure 3.13f). They termed this function *Swish.* Howard *et al.* (2019) approximated Swish by the HardSwish function, which has a very similar shape, but is faster to compute:

$$\text{HardSwish}[z] = \begin{cases} 0 & z < -3 \\ z(z+3)/6 & -3 \leq z \leq 3 \\ z & z > 3 \end{cases}. \tag{3.13}$$

There is no definitive answer as to which of these activations functions is empirically superior, although the leaky ReLU, parameterized ReLU, and many of the continuous functions can be shown to provide minor performance gains over the ReLU in particular situations. We restrict attention to neural networks with the basic ReLU function for the rest of this book because it's easy to characterize the functions that they create in terms of the number of linear regions.

Universal approximation theorem: The *width version* of this theorem states that a network with one hidden layer containing a finite number of hidden units and an activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy. This was proved by Cybenko (1989) for sigmoid activations and was later shown to be true for a larger class of non-linear activation functions (Hornik 1991).

Number of linear regions: Consider a shallow network with $D_i \geq 2$ dimensional inputs and D hidden units. The number of linear regions is determined by the intersections of the D hyperplanes created by the 'joints' in the ReLU functions (*e.g.*, figure 3.8d-f). Each region is created by a different combination of the ReLU functions clipping or not clipping the input. The number of regions created by D hyperplanes in the $D_i \leq D$ dimensional input space was shown by Zaslavsky (1975) to be at most $\sum_{j=0}^{D_i} \binom{D}{j}$. As a rule of thumb, shallow neural networks almost always have a larger number D of hidden units than input dimensions D_i and create a number of linear regions that is between 2^D and 2_i^D.

Problem 3.18

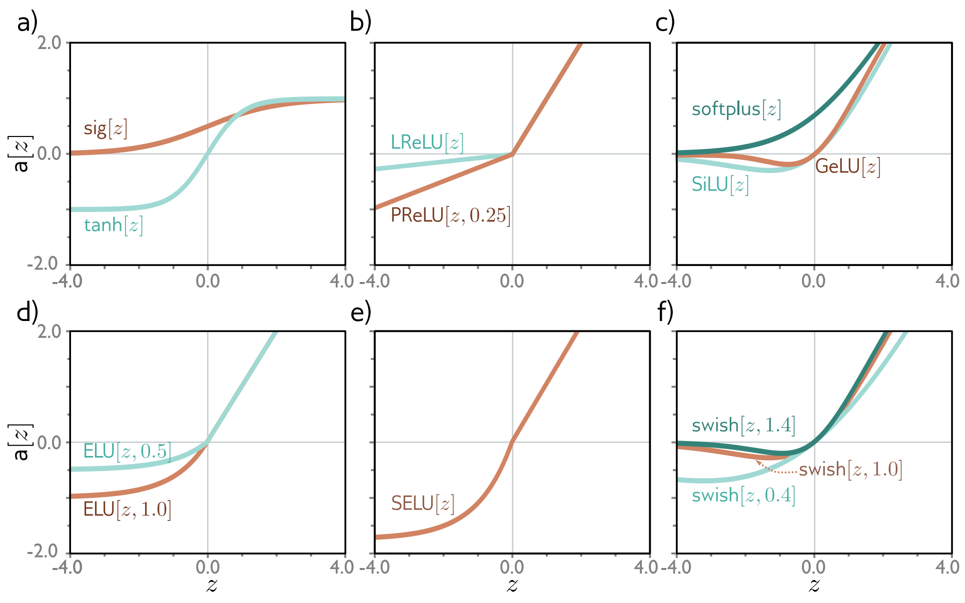


Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Guassian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

Problems

Problem 3.1 What kind of mapping from input to output would be created if the activation function in equation 3.1 was linear so that $a[z] = \psi_0 + \psi_1[z]$? What kind of mapping would be created if the activation function was removed entirely, so $a[z] = z$?

Problem 3.2 For each of the four linear regions in figure 3.3j, indicate which hidden units are inactive and which are active (*i.e.*, which do and do not clip their inputs).

Problem 3.3 Derive expressions for the positions of the 'joints' in function in figure 3.3j in terms of the ten parameters $\boldsymbol{\phi}$. Derive expressions for the slopes of the four linear regions.

Problem 3.4 Draw a version of figure 3.3 where the intercept and slope of the third hidden unit have changed as in figure 3.14c. Assume that the remaining parameters remain the same.

Problem 3.5 Prove that the following property holds for non-negative $z \in \mathbb{R}^+$:

$$\mathrm{ReLU}[\alpha z] = \alpha \mathrm{ReLU}[z]. \tag{3.14}$$

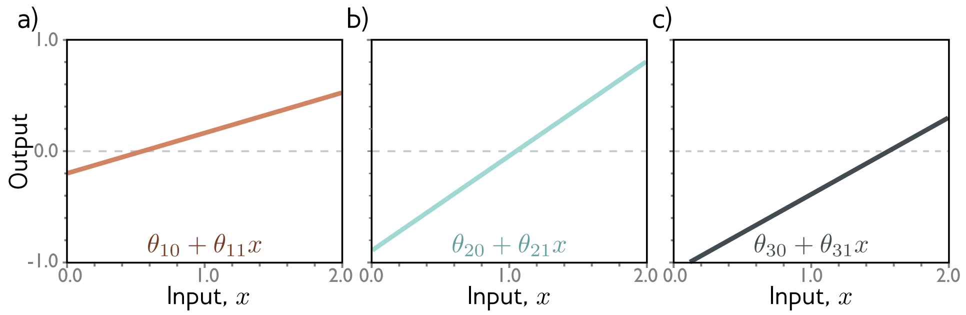


Figure 3.14 Processing in network with one input, three hidden units, and one output for problem 3.4. a-c) The input to each hidden unit is a linear function of the inputs. The first two are the same as in figure 3.3 but the last one has changed.

This is known as the *non-negative homogeneity* property of the ReLU function.

Problem 3.6 Following on from problem 3.5, what happens to the shallow network defined in equations 3.3 and 3.4 when we multiply the slopes $\theta_{11}, \theta_{21}, \theta_{31}$ by a positive constant α and divide the slopes ϕ_1, ϕ_2, ϕ_3 by the same parameter α? What happens if α is negative?

Problem 3.7 Consider fitting the model in equation 3.1 to real data using a least squares loss function. Does this loss function have a single unique minimum? In other words, is there is a single 'best' set of parameters?

Problem 3.8 Consider replacing the ReLU activation function with (i) the Heaviside step function heaviside$[z]$, (ii) the hyperbolic tangent function tanh$[z]$, (iii) the rectangular function rect$[z]$, and (iv) the sinusoidal function sin$[z]$, where:

$$\text{heaviside}[z] = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}, \tag{3.15}$$

and

$$\text{rect}[z] = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}. \tag{3.16}$$

Redraw a version of figure 3.3 for each of these functions. Provide an informal description of the family of functions can be created by neural networks with one input, three hidden units, and one output for each activation function.

Problem 3.9 Show that for figure 3.3, the third linear region has a slope that is the sum of the slopes of first and fourth linear regions.

Problem 3.10 Consider a neural network with one input, one output, and three hidden units. The construction in figure 3.3 shows how this creates four linear regions. Under what circumstances could this network produce a function with fewer than four linear regions?

Problem 3.11 How many parameters does the model in figure 3.6 have?

Problem 3.12 How many parameters does the model in figure 3.7 have?

Problem 3.13 What is the activation pattern for each of the seven regions in figure 3.7? In other words, which hidden units are active (pass the input) and which are inactive (clip the input) for each region?

Problem 3.14 Write out the equations that define the network in figure 3.11. There should be three equations to compute the three hidden units from the inputs and two equations to compute the outputs from the hidden units.

Problem 3.15 What is the maximum possible number of 3D linear regions that can be created by the network in figure 3.11?

Problem 3.16 Write out the equations for a network with two inputs, four hidden units, and three outputs. Draw this model in the style of figure 3.11.

Problem 3.17 Equations 3.11 and 3.12 define a general neural network with D_i inputs, one hidden layer containing D hidden units, and D_o outputs. Find an expression for the number of parameters in the model in terms of D_i, D, and D_o.

Problem 3.18 Show that the maximum number of regions created by a shallow network with $D_i = 2$ dimensional input, $D_o = 1$ dimensional output and $D = 3$ hidden units is seven as in figure 3.8j. Use the result of Zaslavsky (1975) that the maximum number of regions created by partitioning a D_i-dimensional space with D hyperplanes is $\sum_{j=0}^{D_i} \binom{D}{j}$. What is the maximum number of regions if we add two more hidden units to this model so $D = 5$?

Chapter 4

Deep neural networks

The last chapter described shallow neural networks, which have a single hidden layer. This chapter introduces deep neural networks, which have more than one hidden layer. With ReLU activation functions, both shallow and deep networks describe piecewise linear mappings from input to output.

As the number of hidden units increases, shallow neural networks improve their descriptive power. With enough hidden units, they can describe arbitrarily complex functions in high dimensions. However, it turns out that for some functions, the required number of hidden units is impractically large. Deep networks have the advantage that they produce many more linear regions than shallow networks for a given number of parameters, and so from a practical standpoint they can be used to describe a broader family of functions.

4.1 Composing neural networks

To gain insight into the behavior of deep neural networks, we first consider composing two shallow networks, so the output of the first becomes the input to the second. Consider two shallow networks with three hidden units each (figure 4.1a). The first network takes an input x and returns output y and is defined by:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10}+\theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20}+\theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30}+\theta_{31}x], \end{aligned} \tag{4.1}$$

and

$$y = \phi_0+\phi_1h_1+\phi_2h_2+\phi_3h_3. \tag{4.2}$$

The second network takes y as input and returns y' and is defined by:

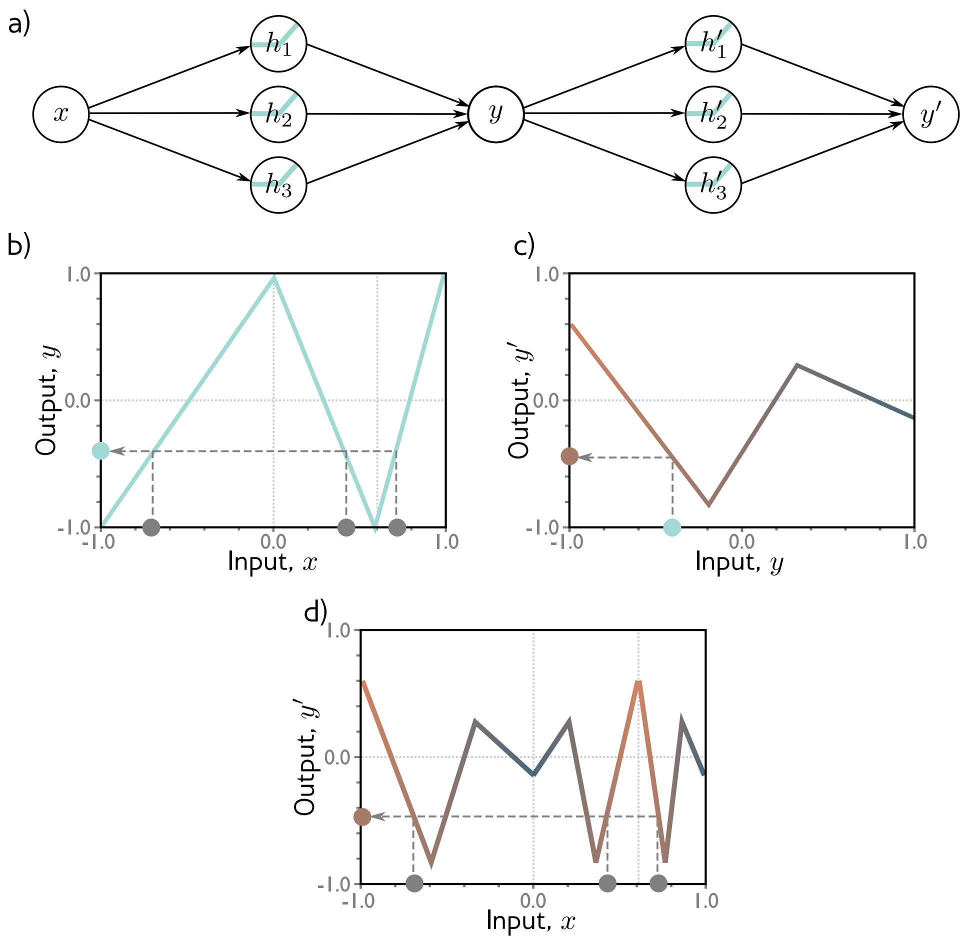


Figure 4.1 Composing two single-layer networks with three hidden units each. a) The output y of the first network constitutes the input to the second network. b) Imagine that the first network maps inputs $x \in [0,1]$ to outputs $y \in [0,1]$ using a function comprised of three linear regions. Let's assume that the function is chosen so that these regions alternate the sign of their slope. Multiple inputs x (gray circles) map to the same output y (cyan circle). c) The second network defines a function comprising three linear regions that takes y and returns y' (*i.e.*, the cyan circle is mapped to the brown circle). d) The combined effect of these two functions when composed is that multiple inputs x (gray circles) are mapped to the same y' (brown circle); the overall result is that the function defined in (c) is duplicated three times, variously flipped and re-scaled.

$$\begin{aligned} h'_1 &= \mathrm{a}[\theta'_{10} + \theta'_{11}y] \\ h'_2 &= \mathrm{a}[\theta'_{20} + \theta'_{21}y] \\ h'_3 &= \mathrm{a}[\theta'_{30} + \theta'_{31}y], \end{aligned} \tag{4.3}$$

and

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3. \tag{4.4}$$

With ReLU activations, this model also describes a family of piecewise linear functions. However, the number of linear regions is potentially greater than for a shallow network with six hidden units. To see this, consider choosing the first network to produce three alternating regions of positive and negative slope (figure 4.1b). This means that three different values of x are mapped to the same output y and the mapping from y to y' is applied three times. The overall effect is that the function defined by the second network is duplicated three times to create nine linear regions. The same principle applies in higher dimensions (figure 4.2).

Problem 4.1

A different way to think about composing networks is that the first network 'folds' the input space x back onto itself so that multiple inputs now generate the same output. Then the second network applies a function, which is replicated at all points that were folded on top of one another (figure 4.3).

4.2 From composing networks to deep networks

The previous section showed that we could create complex functions by passing the output of one shallow neural network into a second network. We'll now show that this is a special case of a deep network with two hidden layers.

The output of the first network ($y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$) is linear and so are the first operations of the second network (equation 4.3 in which we calculate $\theta'_{10} + \theta'_{11}y$, $\theta'_{20} + \theta'_{21}y$, and $\theta'_{30} + \theta'_{31}y$). Applying one linear function to another yields another linear function. Hence, when we substitute the expression for y into equation 4.3 the result is:

$$\begin{aligned} h'_1 &= \mathrm{a}[\theta'_{10} + \theta'_{11}y] &&= \mathrm{a}[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\ h'_2 &= \mathrm{a}[\theta'_{20} + \theta'_{21}y] &&= \mathrm{a}[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\ h'_3 &= \mathrm{a}[\theta'_{30} + \theta'_{31}y] &&= \mathrm{a}[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3], \end{aligned} \tag{4.5}$$

which we can rewrite as:

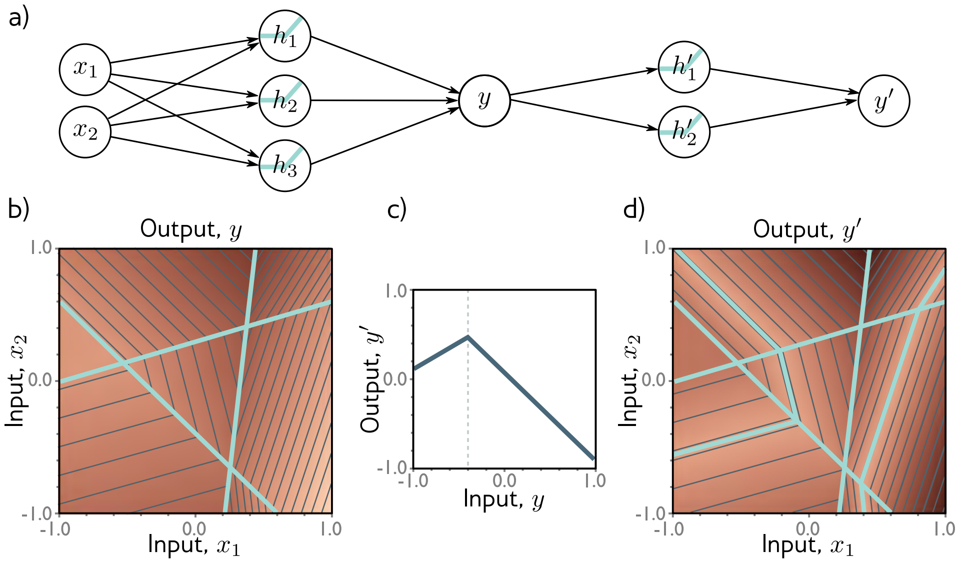


Figure 4.2 Composing neural networks with a 2D input. a) The first network (from figure 3.8) has three hidden units and takes two inputs x_1 and x_2 and returns a scalar output y. This is passed into a second network with two hidden units to produce y'. b) The first network produces a function consisting of seven linear regions, one of which is flat. c) The second network defines a function comprising two linear regions in $y \in [-1, 1]$. d) When composed together, each of the six non-flat regions from the first network is divided into two new regions by the second network to create a total of 13 linear regions.

$$
\begin{aligned}
h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_2 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_2 + \psi_{32}h_2 + \psi_{33}h_3], \qquad (4.6)
\end{aligned}
$$

where $\psi_{10} = \theta'_{10} + \theta'_{11}\phi_0, \psi_{11} = \theta'_{11}\phi_1, \psi_{12} = \theta'_{11}\phi_2$ and so on. The result is a network with two hidden layers (figure 4.4).

It follows that a network with two layers can represent the family of functions created by passing the output of one single-layer network into another. In fact, it represents a broader family, because in equation 4.6, the nine slope parameters $\psi_{11}, \psi_{21}, \ldots \psi_{33}$ can take arbitrary values, whereas in equation 4.5, these parameters are constrained to be the outer product $[\theta'_{11}, \theta'_{21}, \theta'_{31}]^T[\phi_1, \phi_2, \phi_3]$.

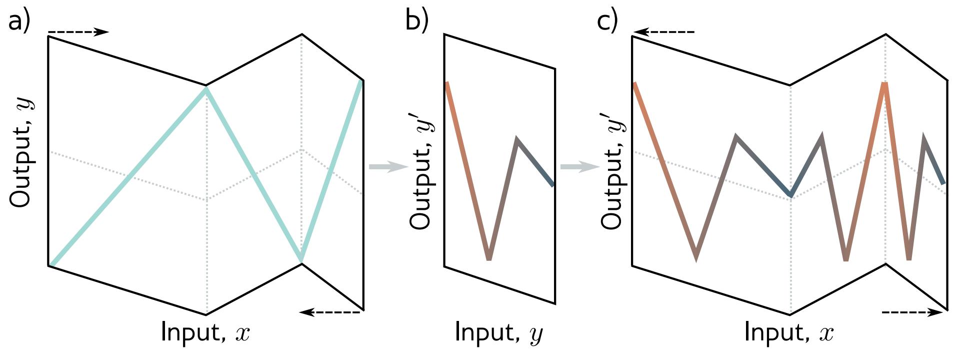


Figure 4.3 Deep networks as folding input space. a) One way to think about the first network from figure 4.1 is that it 'folds' the input space back on top of itself. b) The second network applies its function to the folded space. c) The final output is revealed by 'unfolding' again.

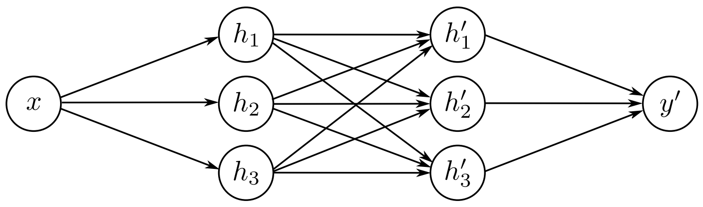


Figure 4.4 Neural network with one input, one output, and two hidden layers, each containing three hidden units.

4.3 Deep neural networks

In the previous section we showed that when we compose two shallow networks, we get a special case of a deep network with two hidden layers. Now let's consider the general case of a deep network with two hidden layers, each containing three hidden units (figure 4.4). The first layer is defined by:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{4.7}$$

the second layer by:

$$\begin{aligned}
h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3], \qquad (4.8)
\end{aligned}$$

and the output by:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3. \qquad (4.9)$$

Considering these equations leads to a different way to think about how the network constructs an increasingly complicated function (figure 4.5):

1. The three hidden units h_1, h_2, and h_3 in the first layer are computed as usual by forming linear functions of the input and passing these through ReLU activation functions (equation 4.7).
2. The pre-activations at the second layer are computed by taking three new linear functions of these hidden units (arguments of the activation functions in equation 4.8). At this point, we effectively have a shallow network with three outputs; we have computed three piecewise linear functions with the 'joints' between linear regions in the same places (see figure 3.6).
3. At the second hidden layer, another ReLU function $\mathrm{a}[\bullet]$ is applied to each function (equation 4.8), which clips them and adds new 'joints' to each.
4. The final output is a linear combination of these hidden units (equation 4.9).

In conclusion, we can either think of each layer as 'folding' the input space, or as creating new functions, which are clipped (creating new regions) and then recombined. Both descriptions provide partial insight into how deep neural networks operate. However, it's important not to lose sight of the fact that this is still merely an equation relating input x to output y'. Indeed, we can combine equations 4.7-4.9 to get a single expression:

$$\begin{aligned}
y' = \; & \phi'_0 + \phi'_1 \mathrm{a}\left[\psi_{10} + \psi_{11}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{12}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{13}\mathrm{a}[\theta_{30} + \theta_{31}x]\right] \\
& + \phi'_2 \mathrm{a}[\psi_{20} + \psi_{21}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{23}\mathrm{a}[\theta_{30} + \theta_{31}x]] \\
& + \phi'_3 \mathrm{a}[\psi_{30} + \psi_{31}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{32}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{33}\mathrm{a}[\theta_{30} + \theta_{31}x]], \qquad (4.10)
\end{aligned}$$

although this is admittedly rather difficult to understand.

4.3.1 Hyperparameters

We can extend the deep network construction to more than two hidden layers; modern networks might have more than a hundred layers with thousands of hidden units at each layer. The number of hidden units in each layer is referred to as the

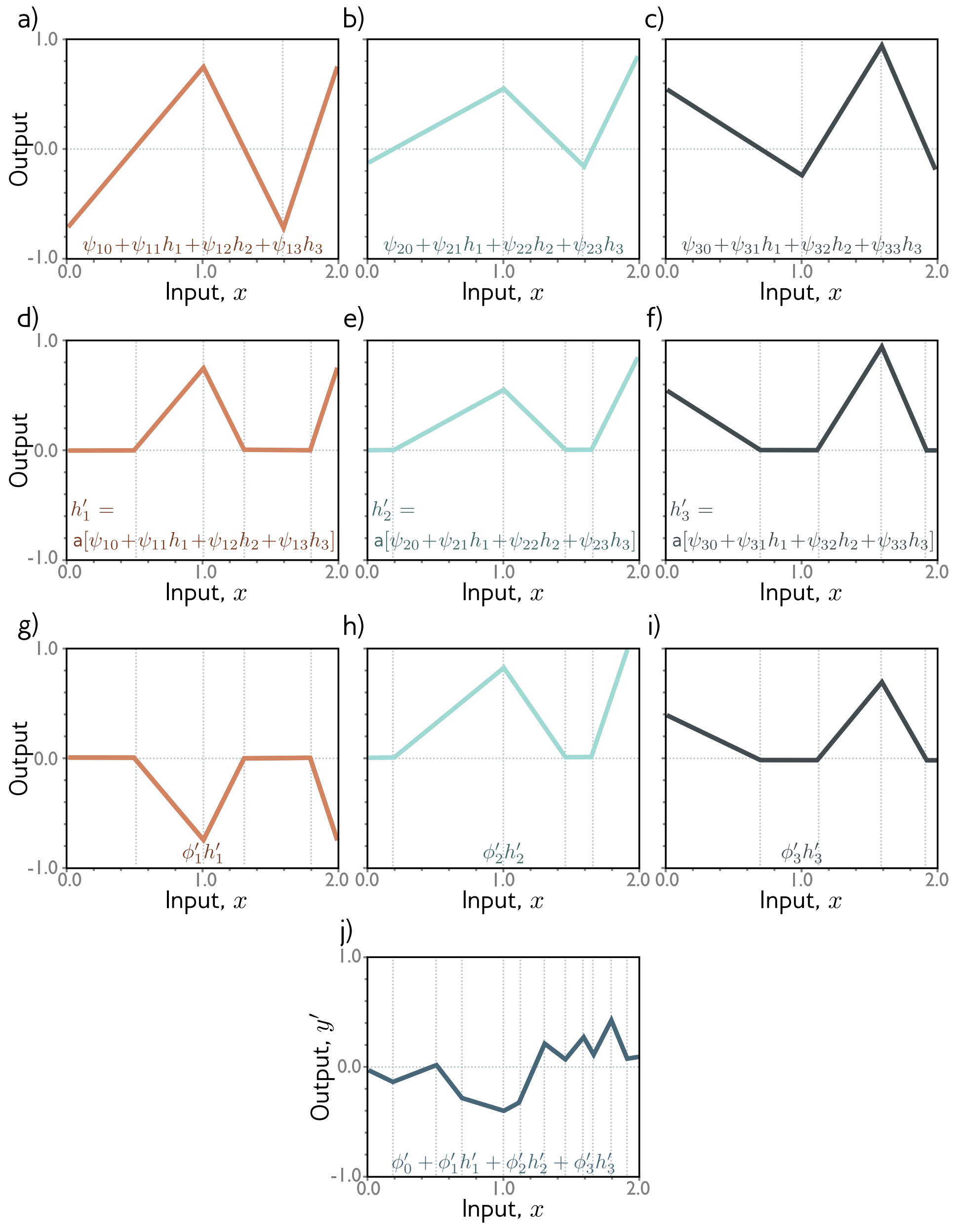


Figure 4.5 Computation for the deep neural network in figure 4.4. a-c) The inputs to the second hidden layer (*i.e.*, the pre-activations at the second layer) are three piecewise linear functions where the 'joints' between the linear regions are at the same places (see figure 3.6). d-f) Each piecewise linear function is clipped to zero by the ReLU activation function. g-i) These clipped functions are then weighted with parameters ϕ'_1, ϕ'_2, and ϕ'_3 respectively. h) Finally, the clipped and weighted functions are summed together and an offset ϕ'_0 that controls the overall height is added.

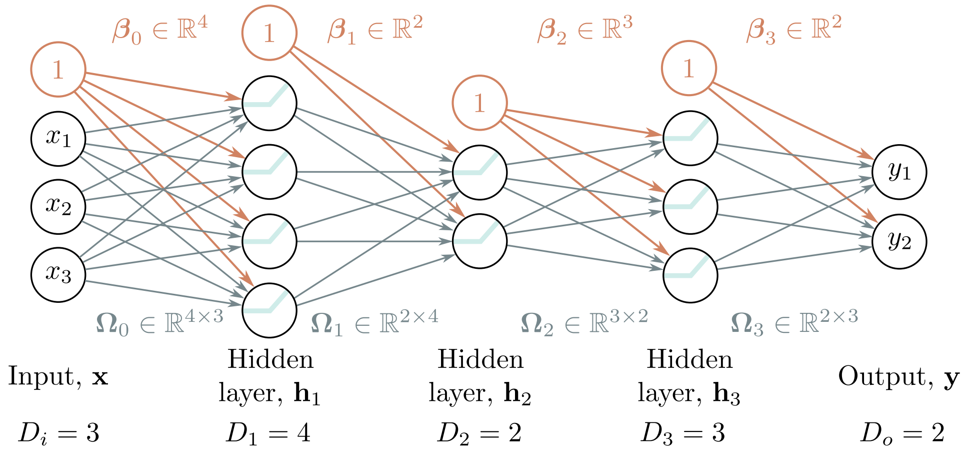

$\boldsymbol{\beta}_0 \in \mathbb{R}^4$
$\boldsymbol{\beta}_1 \in \mathbb{R}^2$
$\boldsymbol{\beta}_2 \in \mathbb{R}^3$
$\boldsymbol{\beta}_3 \in \mathbb{R}^2$
1
x_1
x_2
x_3
y_1
y_2
$\boldsymbol{\Omega}_0 \in \mathbb{R}^{4\times 3}$
$\boldsymbol{\Omega}_1 \in \mathbb{R}^{2\times 4}$
$\boldsymbol{\Omega}_2 \in \mathbb{R}^{3\times 2}$
$\boldsymbol{\Omega}_3 \in \mathbb{R}^{2\times 3}$
Input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output, $\mathbf{y}$
$D_i = 3$
$D_1 = 4$
$D_2 = 2$
$D_3 = 3$
$D_o = 2$

Figure 4.6 Matrix notation for network with $D_i = 3$ dimensional input $\mathbf{x}$, $D_o = 2$ dimensional output $\mathbf{y}$, and $K = 3$ hidden layers $\mathbf{h}_1, \mathbf{h}_2$ and $\mathbf{h}_3$ of dimensions $D_1 = 4$, $D_2 = 2$, and $D_3 = 3$ respectively. The weights are stored in matrices $\boldsymbol{\Omega}_k$ that pre-multiply the activations from the preceding layer to create the pre-activations at the subsequent layer. For example, the weight matrix $\boldsymbol{\Omega}_1$ that computes the pre-activations at $\mathbf{h}_2$ from the activations at $\mathbf{h}_1$ is of dimension 2×4. It is applied to the four hidden units in layer one and creates the two inputs to the hidden units at layer two. The biases are stored in vectors $\boldsymbol{\beta}_k$ and have the dimension of the layer that they feed into. For example, the bias vector $\boldsymbol{\beta}_2$ is length three.

width of the network, and the number of hidden layers as the *depth*. The total number of hidden units is a measure of the *capacity* of the network.

We denote the number of layers as K and the number of hidden units in each layer as $D_1, D_2, \ldots, D_K$. These are examples of *hyperparameters*. They are quantities that are chosen before we learn the parameters of the model (*i.e.*, the slope and intercept terms). For fixed hyperparameters (*e.g.*, $K = 2$ layers with $D_k = 3$ hidden units in each), the model describes a family of functions, and the parameters determine the particular function. Hence, when we also consider the hyperparameters, we can think of neural networks as representing a family of families of functions relating input to output.

4.4 Matrix notation

We have seen that a deep neural network simply consists of linear transformations alternating with activation functions. We could equivalently describe equations 4.7-4.9 in matrix notation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right], \tag{4.11}$$

$$\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{32} & \psi_{32} & \psi_{33} \end{bmatrix}\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right], \tag{4.12}$$

and

$$y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix}\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix}, \tag{4.13}$$

or even more compactly in matrix notation as:

$$\begin{aligned} \mathbf{h} &= \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\Theta}x\right] \\ \mathbf{h}' &= \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right] \\ y &= \boldsymbol{\phi}'_0 + \boldsymbol{\Phi}'\mathbf{h}', \end{aligned} \tag{4.14}$$

where in each case the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of its input.

4.4.1 General formulation

This notation becomes cumbersome for networks with many layers, and so from now on we will describe the vector of hidden units at layer k as $\mathbf{h}_k$, the vector of biases (intercepts) that contribute to hidden layer $k+1$ as $\boldsymbol{\beta}_k$, and the weights (slopes) that are applied to the k^{th} layer and contribute to the $(k+1)^{th}$ layer as $\boldsymbol{\Omega}_k$. A general deep network $\mathbf{y} = \mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with K layers can now be written as:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\ \mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K, \end{aligned} \tag{4.15}$$

The parameters $\boldsymbol{\phi}$ of this model comprise all of these weight matrices and bias vectors $\boldsymbol{\phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k=0}^{K}$.

If the k^{th} layer has D_k neurons, then the bias vector $\boldsymbol{\beta}_{k-1}$ will be of size D_k. The last bias vector $\boldsymbol{\beta}_K$ has the size D_o of the output. The first weight matrix $\boldsymbol{\Omega}_0$ has size $D_1 \times D_i$ where D_i is the size of the input. The last weight matrix $\boldsymbol{\Omega}_K$ is $D_o \times D_K$, and the remaining matrices $\boldsymbol{\Omega}_k$ are $D_k \times D_{k-1}$ (figure 4.6).

Problems 4.2-4.3

We can equivalently write the network as a single function:

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{a}\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{a}\left[\ldots \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{a}\left[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}\right]\right]\ldots\right]\right]. \tag{4.16}$$

4.5 Shallow vs. deep neural networks

In the previous chapter we discussed shallow networks (which have a single hidden layer), and here we have considered deep networks (which have multiple hidden layers). We now compare these models.

4.5.1 Ability to approximate different functions

In section 3.2 we argued that shallow neural networks can model any continuous function arbitrarily closely with enough capacity (hidden units). In this chapter, we saw that a deep network with two hidden layers can represent the composition of two shallow networks. If the first of these shallow networks has enough capacity and the second just computes the identity function, then this deep network can also approximate any continuous function arbitrarily closely. Both shallow and deep networks can approximate any function using piecewise linear regions.

Problem 4.4

4.5.2 Number of linear regions per parameter

Problems 4.5-4.8

A shallow network with one input, one output, and $D > 2$ hidden units can create up to $D+1$ linear regions with non-zero slopes and is defined by $3D+1$ parameters. A deep network with K layers of $D > 2$ hidden units can create a function with up to $(D+1)^K$ linear regions using $3D + 1 + (K-1)D(D+1)$ parameters.

Figure 4.7a shows how the maximum number of linear regions increases as a function of the number of parameters for networks mapping scalar input x to scalar output y. Deep neural networks create much more complex functions for a fixed parameter budget. This effect is magnified as the number of input dimensions D_i increases (figure 4.7b), although computing the maximum number of regions is less straightforward.

This seems attractive, but the flexibility of the functions is still limited by the number of parameters. Both shallow and deep networks can create extremely large numbers of linear regions, but these contain complex dependencies and symmetries.

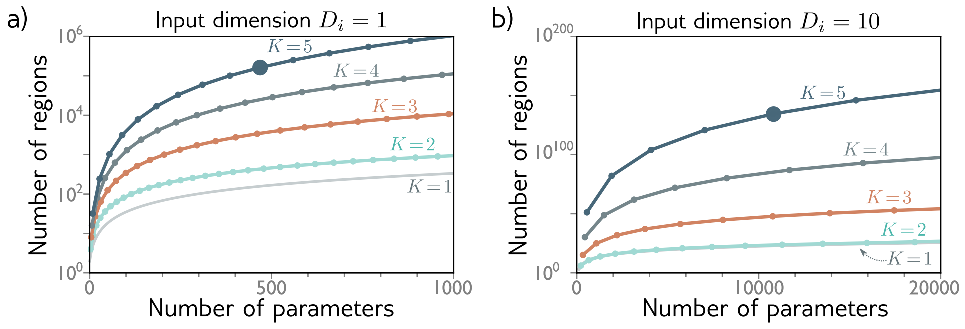


Figure 4.7 The maximum number of linear regions for neural networks increases very rapidly with the network depth. a) Network with $D_i = 1$ input. Each curve represents a fixed number of hidden layers K, as we vary the number of hidden units D per layer. For a fixed parameter budget (horizontal position), deeper networks produce more linear regions than shallower ones. A network with $K = 5$ layers and $D = 10$ hidden units per layer has 471 parameters (highlighted point) and can produce 161,051 regions. b) Network with $D_i = 10$ inputs. Here, a model with $K = 5$ layers and $D = 50$ hidden units per layer has 10,801 parameters (highlighted point) and can create more than 10^{134} linear regions.

We glimpsed some of these when we considered deep networks as 'folding' the input space (figure 4.3). So, it's not clear that the much larger number of regions is an advantage unless (i) there are similar symmetries in the real-world functions that we wish to approximate, or (ii) we have reason to believe that the mapping from input to output really does involve a composition of simpler functions.

4.5.3 Depth efficiency

Although both deep and shallow networks can model arbitrary functions, some functions can be approximated much more efficiently with deep networks. Functions have been identified that require a shallow network with exponentially more hidden units to achieve an equivalent approximation to a deep network. This phenomenon is referred to as the *depth efficiency* of neural networks. This property also seems attractive, but of course, it's still not clear that the real-world functions that we want to approximate fall into this category.

4.5.4 Large, structured inputs

We have discussed fully connected networks where every element of each layer contributes to every element of the subsequent one. However, these are not practical for large, structured inputs like images where the input might comprise $\sim 10^6$ pixels. The number of parameters would be prohibitive, and moreover, we want different parts of the image to be processed similarly; there is no point in independently learning to recognize the same object at every possible position in the image.

The solution is to process local image regions in parallel and then gradually integrate information from increasingly large regions. This kind of local-to-global processing is difficult to specify without using multiple layers (see chapter 10).

4.5.5 Training and generalization

A further reason why deep networks might be advantageous is the ease of fitting; it is easier to train moderately deep networks than shallow ones, although as depth increases further training becomes more difficult again. Deep networks also seem to generalize to new data better than shallow ones. These phenomena are not well understood, and we return to them in chapter 19. In practice, the best results for most tasks have been achieved using networks with tens or hundreds of layers. A rare exception is networks for processing graphs (see chapter 13) where shallower architectures seem to perform better.

4.6 Summary

In this chapter, we first considered what happens when we compose two shallow networks. We argued that the first network 'folds' the input space and the second network then applies a piecewise linear function. The effects of the second network are duplicated where the input space is folded onto itself.

We then showed that this composition of shallow networks is a special case of a deep network with two layers. We interpreted the ReLU functions in each layer as clipping the input functions in multiple places, hence creating more and more 'joints' in the output function. We introduced the idea of hyperparameters, which for neural networks comprise the number of hidden layers and the number of hidden units in each.

Finally, we compared shallow and deep networks. We saw that (i) both networks can approximate any function given enough capacity, (ii) deep networks produce many more linear regions per parameter, (iii) some functions can be approximated much more efficiently by deep networks, (iv) large, structured inputs like images are best processed in a series of stages, and (v) in practice, the best results for most tasks are achieved using deep networks with many layers.

Now that we understand deep and shallow network models, we turn our attention to training them. In the next chapter, we discuss loss functions. For any given parameter values $\boldsymbol{\phi}$, the loss function returns a single number that indicates the mismatch between the model outputs and the ground truth predictions for a training dataset. In chapters 6 and 7, we deal with the training process itself, in which we seek the parameter values that minimize this loss.

Notes

Deep learning: It has long been understood that it is possible to build more complex functions by composing shallow neural networks or developing networks with more than one hidden layer. Indeed, the term 'deep learning' was first used by Dechter (1986). However, interest was limited due to practical concerns; it was not possible to train such networks well. The modern era of deep learning was kick-started by startling improvements in image classification reported by Krizhevsky *et al.* (2012). This sudden progress was arguably due to the confluence of four factors: larger training datasets, improved processing power for training, the use of the ReLU activation function, and the use of stochastic gradient descent (see chapter 6). LeCun *et al.* (2015) present an overview of early advances in the modern era of deep learning.

Number of linear regions: For deep networks using a total of D hidden units with ReLU activations, the upper bound on the number of regions is 2^D (Montúfar *et al.* 2014). The same authors show that a deep ReLU network with D_i dimensional input, K layers each containing $D \geq D_i$ hidden units has $\Omega\left((D/D_i)^{(K-1)D_i} D^{D_i}\right)$ linear regions. Montúfar (2017), Arora *et al.* (2016) and Serra *et al.* (2018) all provide tighter upper bounds that consider the possibility that each layer has different numbers of neurons. Serra *et al.* (2018) provide an algorithm that counts the number of linear regions in a neural network, although it is only practical for very small networks.

If the number of hidden units D in each of the K layers is the same, and D is an integer multiple of the input dimensionality D_i, then the maximum number of linear regions N_r can be computed exactly and is:

$$N_r = \prod_{k=1}^{K-1}\left(\frac{D}{D_i}+1\right)^{D_i} \sum_{j=0}^{D_i}\binom{D}{j}. \tag{4.17}$$

The first term in this expression corresponds to the first $K-1$ layers of the network, which can be thought of as repeatedly folding the input space. However, we now need to devote D/D_i neurons to each input dimension to create these folds. The last term in this equation is the number of regions that can be created by a shallow network and is attributable to the last layer. For further information consult Montúfar *et al.* (2014), Pascanu *et al.* (2014), and Montúfar (2017).

Universal approximation theorem: We argued in section 4.5.1 that if the layers of a deep network have enough hidden units, then the width version of the universal approximation theorem applies: the network can approximate any continuous function on a compact subset of $\mathbb{R}^{D_i}$ to arbitrary accuracy. Lu *et al.* (2017) proved that a network with ReLU activation functions and at least $D_i + 4$ hidden units in each layer can approximate any D_i-dimensional Lebesgue integrable function to arbitrary accuracy given enough layers. This is known as the *depth version* of the universal approximation theorem.

Depth efficiency: There are several results that show that there are classes of functions that can be realized by deep networks but not by any shallow network whose capacity is bounded above exponentially. In other words, it would take an exponentially larger number of units in a shallow network to describe these functions accurately. This is known as the *depth efficiency* of neural networks.

Telgarsky (2016) shows that for any integer k, it is possible to construct networks with one input, one output, and $\mathcal{O}[k^3]$ layers of constant width, which cannot be realized with $\mathcal{O}[k]$ layers and less than 2^k width. Perhaps surprisingly, Eldan & Shamir (2015) showed that when there are multivariate inputs, there is a three-layer network that cannot be realized by any two-layer network if the capacity is sub-exponential in the input dimension. Cohen *et al.* (2015), Safran & Shamir (2016), and Poggio *et al.* (2016) also demonstrate functions that deep networks can approximate efficiently, but shallow ones cannot.

Width efficiency: Lu *et al.* (2017) pose the question of whether there are there wide shallow networks (*i.e.*, shallow networks with lots of hidden units) that cannot be realized by narrow networks whose depth is not substantially larger. They show that there exist classes of wide, shallow networks can only be expressed by narrow networks with polynomial depth. This is known as the *width efficiency* of neural networks. This polynomial lower bound on width is less restrictive than the exponential lower bound on depth, suggesting that depth is more important.

Problems

Problem 4.1 Consider composing the two neural networks in figure 4.8. Draw a plot of the relationship between the input x and output y' for $x \in [-1, 1]$.

Problem 4.2 Using the non-negative homogeneity property of the ReLU function (see problem 3.5), show that:

$$\text{ReLU}\left[\boldsymbol{\beta}_2 + \lambda_2 \cdot \boldsymbol{\Omega}_2 \left[\boldsymbol{\beta}_1 + \lambda_1 \boldsymbol{\Omega}_1 \mathbf{x}\right]\right] = \lambda_0 \cdot \lambda_1 \cdot \text{ReLU}\left[\frac{1}{\lambda_0 \cdot \lambda_1}\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \left[\frac{1}{\lambda_0}\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}\right]\right], \tag{4.18}$$

where λ_1 and λ_2 are non-negative scalars. From this, we see that the weight matrices can be re-scaled by any magnitude as long as the biases are also adjusted, and the scale factors can be re-applied at the end of the network.

Problem 4.3 Write out the equations for a deep neural network that takes $D_i = 5$ inputs, $D_o = 4$ outputs and has three hidden layers of sizes $D_1 = 20$, $D_2 = 10$, and $D_3 = 7$ respectively in both the forms of equations 4.15 and 4.16. What are the sizes of each weight matrix $\boldsymbol{\Omega}_\bullet$ and bias vector $\boldsymbol{\beta}_\bullet$?

Problem 4.4 Choose values for the parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$ for the shallow neural network in equation 3.1 that will define an identity function over a finite range $x \in [a, b]$.

Problem 4.5 Figure 4.9 shows the activations in the three hidden units of a shallow network (as in figure 3.3). The slopes in the hidden units are 1.0, 1.0, and -1.0 respectively and the 'joints' in the hidden units are at positions 1/6, 2/6, and 4/6. Find values of $\phi_0, \phi_1, \phi_2, \phi_3$ that will combine the hidden unit activations as $\phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$ to create a function with four linear regions that oscillate between output values of zero and one. The slope of the left-most region should be positive, the next one negative, and so on. How many linear regions will we create if we compose this network with itself? How many will we create if we compose it with itself K times?

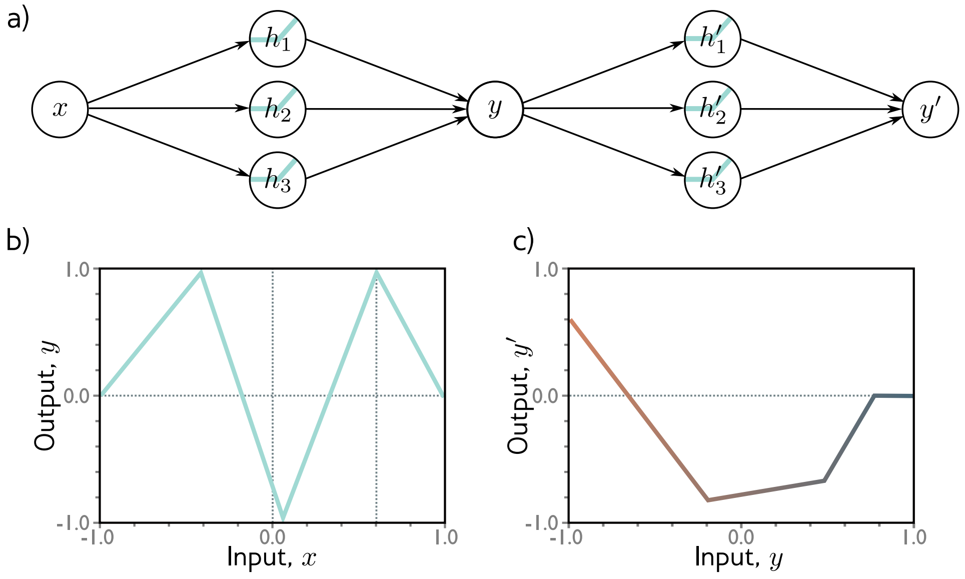


Figure 4.8 Composition of two networks for problem 4.1. a) Two networks are composed together so that the output y of the first network becomes the input to the second. b) The first network computes this function with output values $y \in [-1, 1]$. c) The second network computes this function on the input range $y \in [-1, 1]$.

Problem 4.6 Following from problem 4.5, is it possible to create a function with three linear regions that oscillates back and forth between output values of zero and one using a shallow network with two hidden units? Is it possible to create a function with five linear regions that oscillates in the same way using a shallow network with four hidden units?

Problem 4.7 Consider a deep neural network with a single input, a single output, and K hidden layers each of which contains D hidden units. Show that this network will have a total of $3D + 1 + (K - 1)D(D + 1)$ parameters.

Problem 4.8 Consider two neural networks that map a scalar input x to a scalar output y. The first network is shallow and has $D = 286$ hidden units. The second is deep and has $K = 10$ layers, each containing $D = 5$ hidden units. How many parameters does each network have? How many linear regions can each network make? Which do you think would run faster?

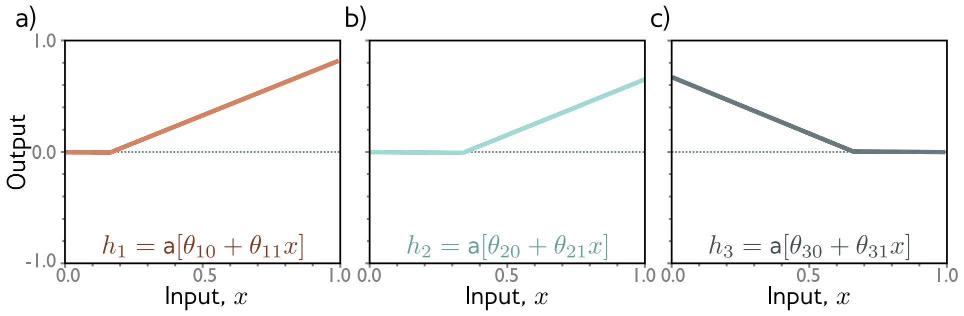


Figure 4.9 Hidden unit activations for problem 4.5. a) First hidden unit has a joint at position $x = 1/6$ and a slope of one in the active region. b) Second hidden unit has a joint at position $x = 2/6$ and a slope of one in the active region. c) Third hidden unit has a joint at position $x = 4/6$ and a slope of minus one in the active region.

Chapter 5

Loss functions

The last three chapters described linear regression, shallow neural networks, and deep neural networks respectively. Each represents a family of functions that map input to output, where the particular member of the family is determined by the model parameters $\boldsymbol{\phi}$. When we train these models, we seek the parameters that produce the best possible mapping from input to output for the task we are considering. This chapter defines what is meant by the 'best possible' mapping.

That definition requires a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. A *loss function* or *cost function* $L[\boldsymbol{\phi}]$ returns a single number that describes the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and their corresponding ground-truth outputs $\mathbf{y}_i$. During training, we seek parameter values $\boldsymbol{\phi}$ that minimize the loss, and hence map the training inputs to the outputs as closely as possible.

We saw one example of a loss function in chapter 2; the least squares loss function is suitable for univariate regression problems for which the target is a scalar $y \in \mathbb{R}$. It computes the sum of the squares of the deviations between the model predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and the true values y_i.

In this chapter, we provide a framework that both justifies the choice of the least squares criterion for scalar outputs, and allows us to build loss functions for other prediction types. We consider *multivariate regression* where the prediction is a vector $\mathbf{y} \in \mathbb{R}^D$, and *classification* where the prediction is one of K categories $y \in [1, \ldots K]$. In the following two chapters, we'll address model training in which the goal is to find the parameter values that minimize these loss functions.

5.1 Maximum likelihood

In this section, we'll develop a recipe for constructing loss functions. Consider a machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that computes an output from input data $\mathbf{x}$. Until now, we have implied that the model directly computes a prediction $\mathbf{y}$. We now shift perspective and consider the model as computing a conditional probability distribution $Pr(\mathbf{y}|\mathbf{x})$ over possible outputs $\mathbf{y}$ given input $\mathbf{x}$.

Appendix B.1.1
Conditional
probability

The loss function encourages each training output $\mathbf{y}_i$ to have high probability or *likelihood* under the distribution $Pr(\mathbf{y}_i|\mathbf{x}_i)$ computed from the corresponding input $\mathbf{x}_i$.

5.1.1 Computing a distribution over outputs

This raises the question of exactly how a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ can be adapted to compute a probability distrifbution. The solution is simple. First, we choose a parametric distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ that is defined on the output domain $\mathbf{y}$. Then we use the network to compute one or more of the distribution parameters $\boldsymbol{\theta}$. [1]

For example, if the prediction domain is $y \in \mathbb{R}$, we might choose the univariate normal distribution, which is defined on $y \in \mathbb{R}$. This distribution is defined by the mean μ and variance σ^2, so $\boldsymbol{\theta} = \{\mu, \sigma^2\}$. In this case, the machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ might predict the mean μ and the variance σ^2 could be treated as an unknown constant.

5.1.2 Maximum likelihood criterion

The model now computes a different distribution parameter $\boldsymbol{\theta}_i$ for each training input $\mathbf{x}_i$. Each observed training output $\mathbf{y}_i$ should have high probability under its corresponding distribution $Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)$. Hence, we choose the model parameters $\boldsymbol{\phi}$ so that they maximize the combined likelihood across all I training examples:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]. \end{aligned} \tag{5.1}$$

For obvious reasons, this is known as the *maximum likelihood* criterion.

Here we are implicitly making two assumptions. First, we assume that the data are identically distributed (the form of the probability distribution over the outputs $\mathbf{y}_i$ is the same for each data point). Second, we assume that the conditional distributions $Pr(\mathbf{y}_i|\mathbf{x}_i)$ of the output given the input are independent, so the total likelihood of the training data decomposes as:

Appendix B.1.2
Independence

$$Pr(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_I|\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_I) = \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i). \tag{5.2}$$

[1] We refer to the parameters of this distribution as *distribution parameters* and denote them by $\boldsymbol{\theta}$ to distinguish them from the model parameters $\boldsymbol{\phi}$.

In other words, we are assuming the data are *independent and identically distributed* or *i.i.d.* for short.

5.1.3 Maximizing log-likelihood

The maximum likelihood criterion in equation 5.1 is not very practical. Each term $Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])$ can be small, and so when we take the product of many of these terms, the result can be an extremely small number. It may be difficult to represent this quantity with finite precision math. Fortunately, we can equivalently maximize the logarithm of the likelihood:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\mathrm{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right] \\ &= \underset{\boldsymbol{\phi}}{\mathrm{argmax}}\left[\log\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]\right] \\ &= \underset{\boldsymbol{\phi}}{\mathrm{argmax}}\left[\sum_{i=1}^{I} \log\left[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]\right]. \end{aligned} \tag{5.3}$$

This *log-likelihood* criterion is equivalent because the logarithm is a monotonically increasing function: if $z > z'$, then $\log[z] > \log[z']$ and vice-versa (figure 5.1). This means that if we change the model parameters $\boldsymbol{\phi}$ in such a way that they improve the log-likelihood criterion, then we are also improving the original maximum likelihood criterion. It also follows that the overall maximum of the two criteria must be in the same place, and so the best model parameters $\hat{\boldsymbol{\phi}}$ are the same in both cases. However, the log-likelihood criterion has the practical advantage that it involves a sum of terms and not a product, and so representing it with finite precision isn't problematic.

5.1.4 Minimizing negative log-likelihood

Finally, we note that by convention, model fitting problems are framed in terms of minimization of a loss. To convert the maximum log-likelihood criterion to a minimization problem, we simply multiply it by minus one, which gives us the *negative log-likelihood criterion*:

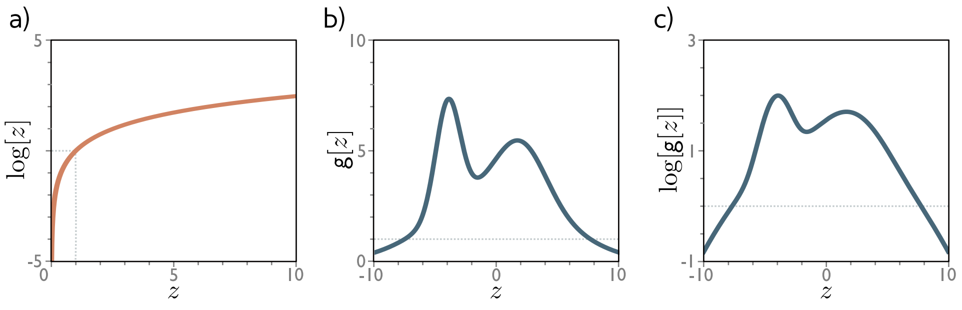


Figure 5.1 Effect of the log transform. a) The log function is monotonically increasing. If $z > z'$, then $\log[z] > \log[z']$. It follows that the maximum of any function $\text{g}[z]$ will be at the same position as the maximum of $\log[\text{g}[z]]$. b) A function $\text{g}[z]$. c) The logarithm of this function $\log[\text{g}[z]]$. All positions on $\text{g}[z]$ with a positive slope retain a positive slope after the log transform and all positions with a negative slope retain a negative slope. The position of the maximum is the same before and after the log transform.

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\text{argmax}}\left[\sum_{i=1}^{I}\log\left[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\right]\right]\\ &= \underset{\boldsymbol{\phi}}{\text{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\right]\right]\\ &= \underset{\boldsymbol{\phi}}{\text{argmin}}\left[\text{L}[\boldsymbol{\phi}]\right], \end{aligned} \tag{5.4}$$

which is what finally forms the loss function $\text{L}[\boldsymbol{\phi}]$.

5.1.5 Inference

The network no longer directly predicts the outputs $\mathbf{y}$, but instead determines a probability distribution over possible values of $\mathbf{y}$. When we perform inference, we often want a single point estimate of $\mathbf{y}$ rather than a distribution and so we return:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{argmax}}\left[Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\boldsymbol{\phi}])\right]. \tag{5.5}$$

It is usually possible to find an expression for this in terms of the distribution parameters $\boldsymbol{\theta}$ predicted by the model. For example, in the univariate normal distribution, the maximum occurs at the mean μ.

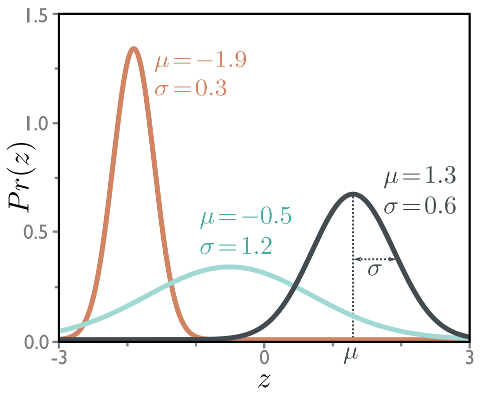


Figure 5.2 The univariate normal distribution (also known as the Gaussian distribution) is defined on the real line $z \in \mathbb{R}$ and has parameters μ and σ^2. The mean μ determines the position of the peak. The positive root of the variance σ^2 (the standard deviation) determines the width of the distribution. Since the total probability density sums to one, the peak becomes higher as the variance decreases and the distribution becomes narrower.

5.2 Recipe for constructing loss functions

Let's summarize the recipe for constructing loss functions for training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using the maximum likelihood approach:

1. Choose a suitable probability distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ that is defined over the domain of the predictions $\mathbf{y}$ and has distribution parameters $\boldsymbol{\theta}$.
2. Set the machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict one or more of these parameters so $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ and $Pr(\mathbf{y}|\boldsymbol{\theta}) = Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}])$.
3. To train the model, find the network parameters $\hat{\boldsymbol{\phi}}$ that minimize the negative log-likelihood loss function over the training dataset pairs $\{\mathbf{x}_i, \mathbf{y}_i\}$:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[L[\boldsymbol{\phi}]\right] = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]\right]. \tag{5.6}$$

4. To perform inference for a new test example $\mathbf{x}$, return either the full distribution $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}])$ or the maximum of this distribution.

We'll devote most of the rest of this chapter to constructing loss functions for common prediction types using this recipe.

5.3 Example 1: univariate regression

We'll start by considering univariate regression models. Here the goal is to predict a single scalar output $y \in \mathbb{R}$ from input $\mathbf{x}$ using a model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$. Following the recipe, we first choose a probability distribution over the output domain y. We'll select the univariate normal distribution (figure 5.2) which is defined over $y \in \mathbb{R}$. This distribution has two parameters (the mean μ and variance σ^2), and has probability density function:

$$Pr(y|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]. \tag{5.7}$$

Second, we set the machine learning model $\mathrm{f}[\mathbf{x},\boldsymbol{\phi}]$ to compute one or more of the parameters of the output distribution. In this case, we will just compute the mean so that $\mu = \mathrm{f}[\mathbf{x},\boldsymbol{\phi}]$:

$$Pr(y|\mathrm{f}[\mathbf{x},\boldsymbol{\phi}],\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y-\mathrm{f}[\mathbf{x},\boldsymbol{\phi}])^2}{2\sigma^2}\right]. \tag{5.8}$$

Our goal will be to find the parameters $\boldsymbol{\phi}$ that make the training data $\{\mathbf{x}_i, y_i\}$ most likely given this distribution (figure 5.3). To accomplish this, we choose a loss function $L[\boldsymbol{\phi}]$ based on the negative log-likelihood:

$$\begin{aligned}
L[\boldsymbol{\phi}] &= -\sum_{i=1}^{I}\log\left[Pr(y_i|\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}],\sigma^2)\right] \\
&= -\sum_{i=1}^{I}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\right].
\end{aligned} \tag{5.9}$$

When we train the model, we seek parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

5.3.1 Least squares loss function

Now let's perform some algebraic manipulations on the loss function. We seek:

$$\begin{aligned}
\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\left[-\sum_{i=1}^{I}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]\right] \\
&= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\left[-\sum_{i=1}^{I}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right]-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right] \\
&= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\left[-\sum_{i=1}^{I}-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right] \\
&= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\left[\sum_{i=1}^{I}(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2\right],
\end{aligned} \tag{5.10}$$

where we have removed the first term between the second and third line because it does not depend on $\boldsymbol{\phi}$. We have removed the denominator between the third and fourth lines as this is just a constant positive scaling factor that does not affect the position of the minimum.

The product of these manipulations is the least squares loss function that we originally introduced when we discussed linear regression in chapter 2:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I}(y_i - \text{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2. \tag{5.11}$$

We see that the least squares loss function follows naturally from the assumptions that the prediction errors are (i) independent and (ii) drawn from a normal distribution with mean $\mu = \text{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ (figure 5.3).

5.3.2 Inference

The network no longer directly predicts y but instead predicts the mean $\mu = \text{f}[\mathbf{x}, \boldsymbol{\phi}]$ of the normal distribution over y. When we perform inference, we usually want a single 'best' point estimate $\hat{y}$; the obvious approach is to take the maximum of the predicted distribution:

$$\hat{y} = \underset{y}{\text{argmax}}\left[Pr(y|\mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}])\right]. \tag{5.12}$$

For the univariate normal, the maximum position is determined by the mean parameter μ (figure 5.2). This is exactly what the model computed and so $\hat{y} = \text{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$.

5.3.3 Estimating variance

To formulate the least squares loss function, we assumed that the network predicted the mean of a normal distribution. The final expression in equation 5.11 (perhaps surprisingly) does not depend on the variance σ^2. However, there is nothing to stop us from treating σ^2 as a parameter of the model and minimizing equation 5.9 with respect to both the model parameters $\boldsymbol{\phi}$ and the distribution variance σ^2:

$$\hat{\boldsymbol{\phi}}, \hat{\sigma}^2 = \underset{\boldsymbol{\phi}, \sigma^2}{\text{argmin}}\left[-\sum_{i=1}^{I}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_i - \text{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]\right]. \tag{5.13}$$

In inference, the model predicts the mean $\mu = \text{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$ from the input, and we learned the variance σ^2 during the training process. The former is the best prediction. The latter tells us about the uncertainty of the prediction.

5.3.4 Heteroscedastic regression

The model above assumes that the variance of the data is constant everywhere. However, this might be unrealistic. When the uncertainty of the model varies as a function of the input data, we refer to this as *heteroscedastic* (as opposed to *homoscedastic* where the uncertainty is constant).

A simple way to model this is to train a neural network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that computes both the mean and the variance. For example, consider a shallow network with two

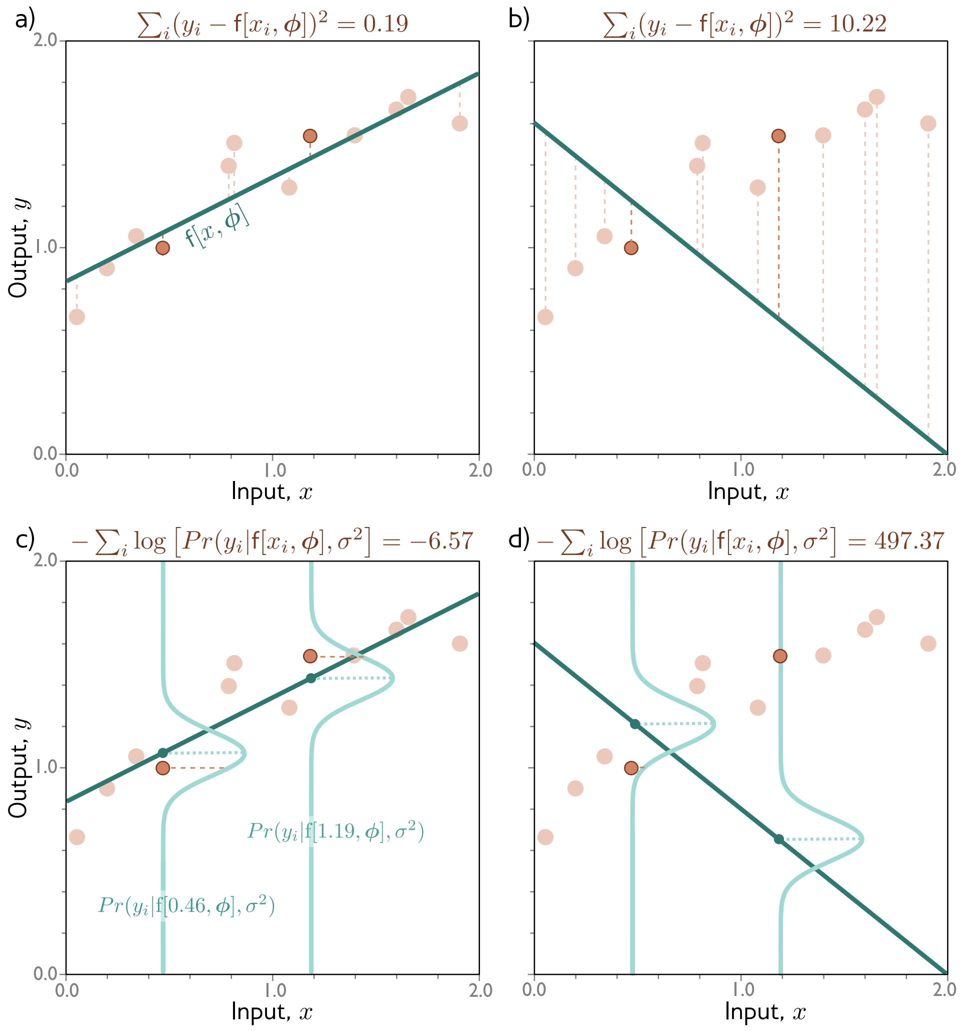


Figure 5.3 Equivalence of least squares and maximum likelihood loss function based on the normal distribution. a) Consider the linear model from figure 2.2. The least squares criterion minimizes the sum of the squares of the deviations (dashed lines) between the model prediction $\mathrm{f}[x_i, \phi]$ (green line) and the true output values y_i (orange dots). Here the fit is good and so these deviations are small (*e.g.*, for the two example highlighted points). b) For this choice of parameters, the fit is bad and so the squared deviations are large. c) This least squares criterion follows from the assumption that the model predicts the mean of a normal distribution over the outputs and that we minimize the negative log probability. For the first case, the model fits well, and so the probability $Pr(y_i|x_i)$ of the observed data (horizontal orange dashed lines) is large (and hence the negative log probability is small). d) For the second case, the model fits badly and so the probability is small (and hence the negative log probability is large).

outputs. We'll denote the first output as $\text{f}_1[\mathbf{x}, \boldsymbol{\phi}]$ and use this to predict the mean and we'll denote the second output as $\text{f}_2[\mathbf{x}, \boldsymbol{\phi}]$ and use it to predict the variance.

There is one complication though; the variance must be positive, but we can't guarantee that the network will always produce a positive output. To ensure that the computed variance is positive, we'll pass the second network output through a function that maps an arbitrary value to a positive one. A suitable choice is the squaring function. So we have:

$$\begin{aligned} \mu &= \text{f}_1[\mathbf{x}, \boldsymbol{\phi}] \\ \sigma^2 &= \text{f}_2[\mathbf{x}, \boldsymbol{\phi}]^2, \end{aligned} \tag{5.14}$$

which results in the loss function:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\text{argmin}} \left[-\sum_{i=1}^{I} \log \left[\frac{1}{\sqrt{2\pi \text{f}_2[\mathbf{x}_i, \boldsymbol{\phi}]^2}} \right] - \frac{(y_i - \text{f}_1[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\text{f}_2[\mathbf{x}_i, \boldsymbol{\phi}]^2} \right]. \tag{5.15}$$

Homoscedastic and heteroscedastic models are compared in figure 5.4.

5.4 Example 2: multivariate regression

In multivariate regression, the goal is to predict a multidimensional target $\mathbf{y} \in \mathbb{R}^{D_o}$ from input data $\mathbf{x}$, where D_o is the number of target dimensions. Following the recipe in section 5.2, we first select a distribution over this domain. One possible choice would be the multivariate normal distribution, which describes the covariance of the prediction errors. However, a simpler approach is just to assume that the errors in each dimension d are independent and use a product of univariate normal distributions:

$$Pr(\mathbf{y}|\boldsymbol{\mu}, \sigma^2) = \prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_d - \mu_d)^2}{2\sigma^2} \right], \tag{5.16}$$

where y_d is the d^{th} entry of the ground truth output $\mathbf{y}$. Each distribution has a different mean μ_d (stored in a vector $\boldsymbol{\mu}$), but they all have the same variance σ^2.

We then use the machine learning model to predict some or all of the parameters of this distribution. Let's assume that we have a neural network model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with D_o outputs $\text{f}_d[\mathbf{x}, \boldsymbol{\phi}]$ that predict the means μ_d, so the likelihood is now:

$$Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}], \sigma^2) = \prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_d - \text{f}_d[\mathbf{x}, \boldsymbol{\phi}])^2}{2\sigma^2} \right]. \tag{5.17}$$

The negative log-likelihood loss function for a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$ is:

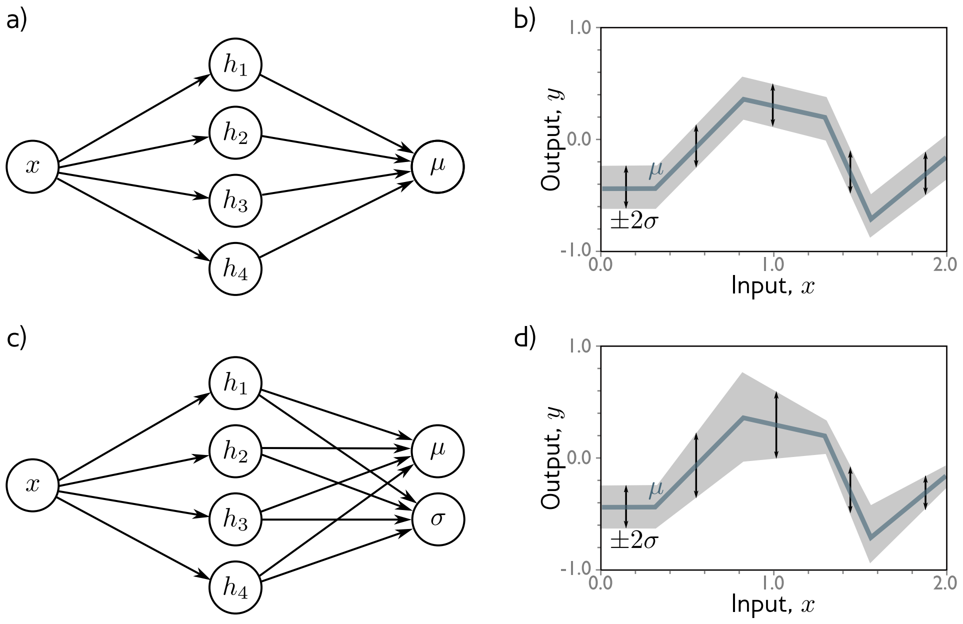


Figure 5.4 Homoscedastic vs. heteroscedastic regression. a) A shallow neural network for homoscedastic regression predicts just the mean μ of the output distribution from the input x. b) The result is that while the mean (blue line) is a piecewise linear function of the input x, the variance is constant everywhere (arrows and gray region show ± 2 standard deviations). c) A shallow neural network for heteroscedastic regression also predicts the variance σ^2 (or more precisely computes its square root, which we then square). d) The result is that the standard deviation also becomes a piecewise linear function of the input x.

$$
\begin{aligned}
\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}],\sigma^2)\right]\right]\\
&= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[\prod_{d=1}^{D_o}\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_{id}-\mathrm{f}_d[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]\right]\\
&= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\sum_{d=1}^{D_o}(y_{id}-\mathrm{f}_d[\mathbf{x}_i,\boldsymbol{\phi}])^2\right], \qquad (5.18)
\end{aligned}
$$

where y_{id} is the d^{th} element of training target $\mathbf{y}_i$, and we have used similar steps as those in equation 5.10 between the second and third lines. This once more results in a least-squares formulation. For a new input $\mathbf{x}$, the prediction for $\mathbf{y}$ is just the vector of means $\boldsymbol{\mu} = \mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$.

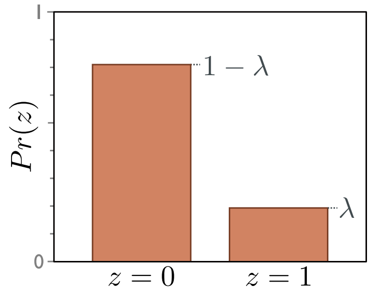


Figure 5.5 Bernoulli distribution. The Bernoulli distribution is defined on the domain $z \in \{0,1\}$ and has a single parameter λ that denotes the probability of observing $z=1$. It follows that the probability of observing $z=0$ is $1-\lambda$.

5.4.1 Estimating variances

We may wish to estimate a vector $\mathbf{y}$ of quantities that are very different in magnitude. For example, consider predicting the height in meters and the weight in kilos of a human being from their age. We would expect the weight to have a much larger variance, simply because of the difference in units. If we use the above criterion, then the network will devote more effort to minimizing errors in the weight than errors in the height.

Problems 5.1-5.3

In this case, one approach is to learn a separate variance σ_d^2 for each dimension as part of the loss function together with the network parameters. However, a commonly used practical alternative is just to scale the output dimensions to have the same empirical variance before training the model and then re-scale the outputs of the model in the inverse way to make the final prediction.

5.5 Example 3: binary classification

In *binary classification*, the goal is to predict which of two discrete classes $y \in \{0,1\}$ the input data $\mathbf{x}$ belongs to. In this context, we refer to y as a *label*. Examples of binary classification include (i) predicting whether a restaurant review is positive ($y=1$) or negative ($y=0$) from text data $\mathbf{x}$ and (ii) predicting whether a tumor is present ($y=1$) or absent ($y=0$) from an MRI scan $\mathbf{x}$.

Once again, we will follow the recipe from section 5.2 to construct the loss function. First, we choose a probability distribution over the output space $y \in \{0,1\}$. A suitable probability distribution is the Bernoulli distribution which is defined on the domain $\{0,1\}$. This has a single parameter $\lambda \in [0,1]$ that represents the probability that y takes the value one (figure 5.5):

$$Pr(y|\lambda) = \begin{cases} 1-\lambda & y=0 \\ \lambda & y=1 \end{cases}, \tag{5.19}$$

which can equivalently be written as

$$Pr(y|\lambda) = (1-\lambda)^{1-y} \cdot \lambda^{y}. \tag{5.20}$$

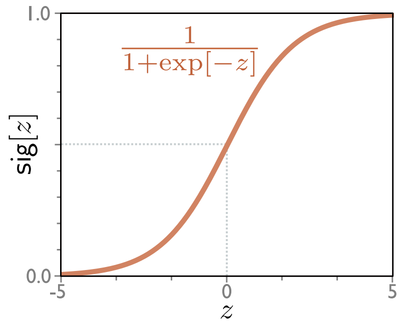


Figure 5.6 Logistic sigmoid function. This function maps the real line $z \in \mathbb{R}$ to numbers between zero and one, and so $\text{sig}[z] \in [0, 1]$. An input of 0 is mapped to 0.5. Negative inputs are mapped to numbers below 0.5 and positive inputs to numbers above 0.5.

Second, we set the machine learning model $\text{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict the single distribution parameter λ. However, λ can only take values in the range $[0, 1]$ and we cannot guarantee that the network output will lie in this range. Consequently, we pass the network output through a function that maps the real line $\mathbb{R}$ to $[0, 1]$. A suitable function is the *logistic sigmoid* (figure 5.6): Problem 5.4

$$\text{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{5.21}$$

Hence, we predict the distribution parameter as $\lambda = \text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]$. The likelihood is now:

$$Pr(y|\mathbf{x}) = (1 - \text{sig}[\text{f}[\mathbf{x}|\boldsymbol{\phi}]])^{1-y} \cdot \text{sig}[\text{f}[\mathbf{x}|\boldsymbol{\phi}]]^{y}. \tag{5.22}$$

This is depicted in figure 5.7 for a shallow neural network model. The loss function is the negative log-likelihood of the training set:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} -(1 - y_i) \log \left[1 - \text{sig}[\text{f}[\mathbf{x}_i|\boldsymbol{\phi}]]\right] - y_i \log \left[\text{sig}[\text{f}[\mathbf{x}_i|\boldsymbol{\phi}]]\right]. \tag{5.23}$$

For reasons that will be explained in section 5.8, this is sometimes known as the *binary cross-entropy loss.*

The transformed model output $\text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]$ predicts the parameter λ of the Bernoulli distribution. This represents the probability that $y = 1$ and it follows that $1 - \lambda$ represents the probability that $y = 0$. When we perform inference, we may want a point estimate of y, and so we set $y = 1$ if $\lambda > 0.5$ and $y = 0$ otherwise. Problem 5.5

5.6 Example 4: multi-class classification

In multi-class classification, the goal is to classify an input data example $\mathbf{x}$ into $K > 2$ classes so $y \in \{1, 2, \ldots, K\}$. Real-world examples include (i) predicting which of $K = 10$ digits y is present in an image $\mathbf{x}$ of a handwritten number, and (ii) predicting which of K possible words y follows an incomplete sentence $\mathbf{x}$.

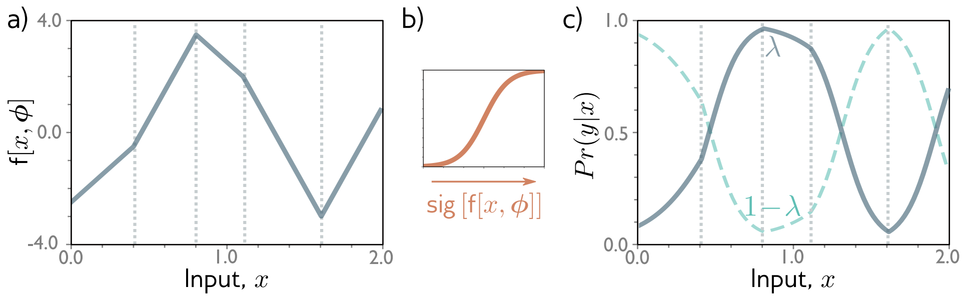


Figure 5.7 Binary classification model a) Output of neural network is a piecewise linear function that can take arbitrary real values. b) This is transformed by the logistic sigmoid function, which compresses these values to the range $[0, 1]$. c) The transformed output predicts the probability λ that $y = 1$ (solid line). The probability that $y = 0$ is hence $1 - \lambda$ (dashed line). For any fixed x (vertical slice) we retrieve the two values of a Bernoulli distribution similar to that in figure 5.5. The loss function favors model parameters that produce large values of λ at positions x_i that are associated with positive examples $y_i = 1$ and small values of λ at positions associated with negative examples $y_i = 0$.

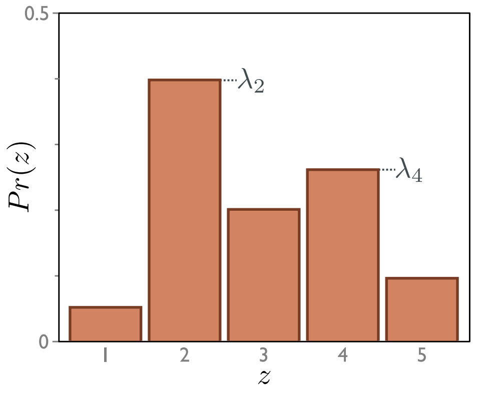


Figure 5.8 Categorical distribution. The categorical distribution assigns probabilities to $K > 2$ categories, with probabilities $\lambda_1, \lambda_2, \ldots, \lambda_K$. In this example, there are five categories and so $K = 5$. To ensure that this is a valid probability distribution, each parameter λ_k must lie in the range $[0, 1]$ and all K parameters must sum to one.

We once more follow the recipe from section 5.2. We first choose a distribution over the prediction space y. In this case, we have $y \in \{1, 2, \ldots, K\}$ so we choose the categorical distribution (figure 5.8), which is defined on this domain. This has K parameters $\lambda_1, \lambda_2, \ldots, \lambda_K$, which determine the probability of each category:

$$Pr(y = k) = \lambda_k. \tag{5.24}$$

Each parameter is constrained to take a value between zero and one and they must collectively sum to one to ensure a valid probability distribution.

Then we use a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with K outputs to compute these K parameters from the input $\mathbf{x}$. Unfortunately, the network outputs will not necessarily obey the

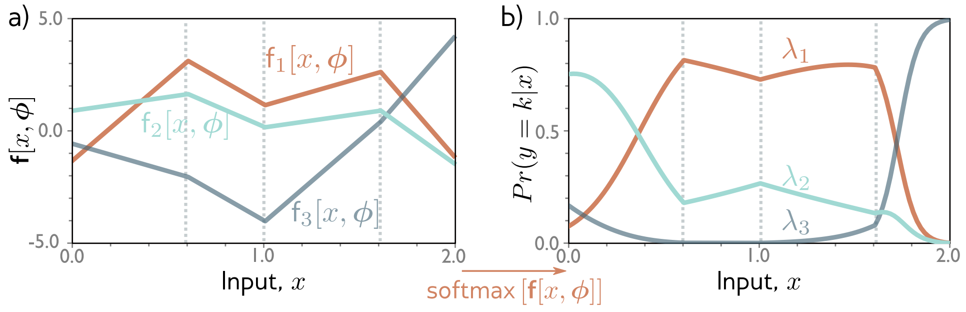


Figure 5.9 Multi-class classification for $K = 3$ classes. a) The neural network has three piecewise linear outputs, which can take arbitrary values. b) After applying the softmax function, these outputs are constrained to be non-negative and to sum to one. Hence, for a given input $\mathbf{x}$, we calculate a valid set of parameters of the categorical distribution. Any vertical slice of this plot produces three values that would form the heights of the bars in a categorical distribution similar to figure 5.8.

aforementioned constraints. Consequently, we pass the K outputs of the network through a function that ensures these constraints are respected. A suitable choice is the *softmax* function (figure 5.9). This takes an arbitrary vector of length K and returns a vector of the same length but where the elements are now in the range $[0, 1]$ and sum to one. The k^{th} output of the softmax function is:

$$\text{softmax}_k[\mathbf{z}] = \frac{\exp[z_k]}{\sum_{k=1}^{K}\exp[z_k]}, \tag{5.25}$$

where the exponential functions ensure positivity, and the sum in the denominator ensures that the K numbers sum to one.

The likelihood that input $\mathbf{x}$ has label y is hence:

$$Pr(y=k|\mathbf{x}) = \text{softmax}_k[\mathbf{f}[\mathbf{x},\boldsymbol{\phi}]]. \tag{5.26}$$

This model is illustrated in figure 5.9. The loss function is the negative log-likelihood of the training data:

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\sum_{i=1}^{I}\log\left[\text{softmax}_{y_i}\left[\mathbf{f}\left[\mathbf{x}_i,\boldsymbol{\phi}\right]\right]\right] \\ &= -\sum_{i=1}^{I}\text{f}_{y_i}\left[\mathbf{x}_i,\boldsymbol{\phi}\right] - \log\left[\sum_{k=1}^{K}\exp\left[\;\text{f}_k\left[\mathbf{x}_i,\boldsymbol{\phi}\right]\right]\right], \end{aligned} \tag{5.27}$$

where $\text{f}_k[\mathbf{x},\boldsymbol{\phi}]$ denotes the k^{th} output of the neural network. For reasons that will be explained in section 5.8, this is known as the *multi-class cross-entropy loss.*

For a given input $\mathbf{x}$, the transformed model output represents a categorical distribution over the possible predictions $y \in \{1, 2, \ldots, K\}$. To get a point estimate, we take the most probable category $\hat{y} = \text{argmax}_k[Pr(y = k|\mathbf{x})]$.

5.7 Predicting other data types

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
symmetric positive definite matrix	$\mathbf{Y} \in \mathbb{R}^{K\times K}$ $\mathbf{z}^T\mathbf{Y}\mathbf{z} > 0 \quad \forall \mathbf{z} \in \mathbb{R}^K$	Wishart	predicting covariances
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multi-class classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots K]$	Plackett-Luce	ranking

Figure 5.10 Distributions for loss functions for different prediction types.

In this chapter, we have focused on regression and classification because these problems are very common. However, to make different types of prediction, we simply choose an appropriate distribution over that domain and apply the recipe in section 5.2. Figure 5.10 enumerates a series of probability distributions and their prediction domain. Some of these are explored in the problems at the end of the chapter.

Problems 5.6-5.10

When we want to make two or more different types of prediction simultaneously, the usual approach is to assume that the errors in each are independent. For

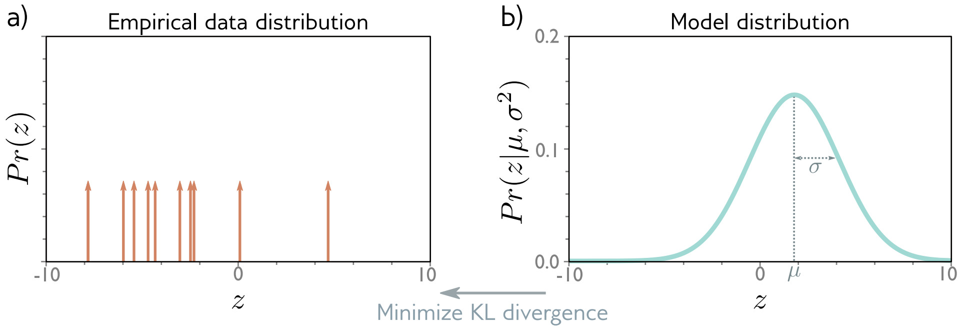


Figure 5.11 Cross-entropy method. a) Empirical distribution of training samples (arrows denote Dirac delta functions). b) Model distribution (a normal distribution with parameters $\boldsymbol{\theta} = \mu, \sigma^2$. In the cross-entropy approach, we minimize the distance (KL divergence) between these two distributions as a function of the model parameters $\boldsymbol{\theta}$.

example, if we want to predict wind direction and strength, then we might choose the von Mises distribution (which is defined on circular domains) for the direction and the exponential distribution (which is defined on positive real numbers) for the strength. The independence assumption implies that the joint likelihood of the two predictions is just the product of individual likelihoods, and these terms will become additive when we compute the negative log-likelihood.

5.8 Cross-entropy loss

In this chapter, we developed loss functions based on minimizing negative log-likelihood. However, it is common in the literature to refer to *cross-entropy* losses. In this section, we'll introduce the cross-entropy loss and show that it is equivalent to using negative log-likelihood.

The cross-entropy loss is based on the idea of finding parameters $\boldsymbol{\theta}$ that minimize the distance between the empirical distribution $q(y)$ of the observed data y and a model distribution $Pr(y|\boldsymbol{\theta})$ (figure 5.11). The 'distance' between two probability distributions $q(z)$ and $p(z)$ can be evaluated using the Kullback-Leibler (KL) divergence:

Appendix B.1.4
KL Divergence

$$\mathrm{KL}[q||p] = \int_{-\infty}^{\infty} q(z)\log[q(z)]dz - \int_{-\infty}^{\infty} q(z)\log[p(z)]dz. \qquad (5.28)$$

Now consider that we observe an empirical distribution of data at points $\{y_i\}_{i=1}^{I}$. We can describe this as a weighted sum of point masses:

$$q(y) = \frac{1}{I}\sum_{i=1}^{I}\delta[y-y_i], \tag{5.29}$$

where $\delta[\bullet]$ is the Dirac delta function . We want to minimize the KL-divergence between the model distribution $Pr(y|\boldsymbol{\theta})$ and this empirical distribution:

Appendix B.1.5
Dirac delta

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[\int_{-\infty}^{\infty} q(y)\log[q(y)]dy - \int_{-\infty}^{\infty} q(y)\log\left[Pr(y|\boldsymbol{\theta})\right]dy\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty} q(y)\log\left[Pr(y|\boldsymbol{\theta})\right]dy\right], \end{aligned} \tag{5.30}$$

where the first term disappears as it has no dependence on $\boldsymbol{\theta}$. The remaining second term is known as the *cross-entropy.* It can be interpreted as the amount of uncertainty that remains in one distribution after taking into account what we already know from the other. Now, we substitute in the definition of $q(y)$ from equation 5.29:

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\theta}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty}\frac{1}{I}\sum_{i=1}^{I}\delta[y-y_i]\log\left[Pr(y|\boldsymbol{\theta})\right]dx\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\frac{1}{I}\sum_{i=1}^{I}\log\left[Pr(y_i|\boldsymbol{\theta})\right]\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(y_i|\boldsymbol{\theta})\right]\right], \end{aligned} \tag{5.31}$$

where we have eliminated the constant scaling factor $1/I$ in the last line as this does not affect the position of the minimum.

In machine learning, the distribution parameters $\boldsymbol{\theta}$ are computed by the model $\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}]$, and so we have:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(y_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\right]\right]. \tag{5.32}$$

This is exactly the negative log-likelihood criterion from the recipe in section 5.2.

It follows that the negative log-likelihood criterion (from maximizing the data likelihood) and the cross-entropy criterion (from minimizing the distance between the model and empirical data distributions) are equivalent.

5.9 Summary

We previously considered neural networks as directly predicting outputs $\mathbf{y}$ from data $\mathbf{x}$. In this chapter, we shifted perspective to think about neural networks as computing the parameters $\boldsymbol{\theta}$ of probability distributions $Pr(\mathbf{y}|\boldsymbol{\theta})$ over the output space. This led to a principled approach to building loss functions. We select model parameters $\boldsymbol{\phi}$ that maximize the likelihood of the observed data under these distributions. We saw that this is equivalent to minimizing the negative log-likelihood.

The least squares criterion for regression is a natural consequence of this approach; it follows from the assumption that y is normally distributed and that we are predicting the mean. We also saw how the regression model can be (i) extended to estimate the uncertainty over the prediction, (ii) extended to make that uncertainty dependent on the input (the heteroscedastic model), and (iii) applied to multivariate predictions $\mathbf{y}$. We applied the same approach to both two-class and multi-class classification and derived loss functions for each. We discussed how to tackle more complex data types. Finally, we argued that cross-entropy is an alternative and equivalent way to think about fitting models.

In previous chapters, we have developed neural network models. In this chapter, we developed loss functions for deciding how well a model describes the training data for a given set of parameters. The next chapter considers model training in which we aim to find the model parameters that minimize this loss.

Notes

Losses based on the normal distribution: Nix & Weigend (1994) and Williams (1996) investigated heteroscedastic non-linear regression in which both the mean and the variance of the output are functions of the input. In the context of unsupervised learning, Burda *et al.* (2016) use a loss function based on a multivariate normal distribution with diagonal covariance, and Dorta *et al.* (2018) use a loss function based on a normal distribution with (approximated) full covariance.

Robust regression: Qi *et al.* (2020) investigate the properties of regression models that minimize mean absolute error rather than mean squared error. This loss function follows from assuming a Laplace distribution over the outputs and estimates the median output for a given input rather than the mean. Barron (2019) presents a novel loss function that parameterizes the degree of robustness. When interpreted in a probabilistic context, it yields a family of univariate probability distributions that includes the normal and Cauchy distributions as special cases.

Estimating quantiles: Sometimes we may not want to estimate the mean or median in a regression task but may instead want to predict a quantile. For example, this would be useful for risk models, where we might want to know that the true value will be less than the predicted value 90% of the time. This is known as *quantile regression* (Koenker & Hallock 2001). This could be done by fitting a heteroscedastic regression model and then estimating the quantile based on the predicted normal distribution. Alternatively, the quantiles can be estimated directly using *quantile loss* (also known as *pinball loss*). In practice, this minimizes the absolute deviations of the data from the model but weights the deviations in one direction more than the other. Recent work has investigated simultaneously predicting multiple quantiles to get an idea of the overall distribution shape (Rodrigues & Pereira 2018).

Class imbalance and focal loss: Lin *et al.* (2017) address the issue of data imbalance in classification problems. If the number of examples in some classes are much greater than in others, then the standard maximum likelihood/cross-entropy loss does not work well; the model may concentrate on becoming more confident about well-classified examples from the dominant classes and classify less well-represented classes poorly. Lin *et al.* (2017) introduce *focal loss*, which adds a single extra parameter that down-weights the effect of well-classified examples and improves performance in this setting.

Learning to rank: Cao *et al.* (2007), Xia *et al.* (2008), and Chen *et al.* (2009) all used the Plackett-Luce model in loss functions for learning to rank data. This is the *listwise* approach to learning to rank as the model ingests an entire list of objects to be ranked at once. Alternative approaches are the *pointwise* approach in which a single object is ingested by the model and the *pairwise* approach where the model ingests pairs of objects. Different approaches for learning to rank are summarized in Chen *et al.* (2009).

Probabilistic models for other data types: Fan *et al.* (2020) use a loss based on a Beta distribution for predicting values between zero and one. Jacobs *et al.* (1991) and Bishop (1994) investigated machine learning models for multimodal data. These model the output as mixture of Gaussians that is conditional on the input and are called *mixture density networks*. Prokudin *et al.* (2018) investigated several models for predicting direction, including one based on the von Mises distribution. Fallah *et al.* (2009) constructed loss functions for prediction counts using the Poisson distribution. Ng *et al.* (2017) used loss functions based on the gamma distribution to predict duration.

Non-probabilistic approaches: It is not strictly necessary to adopt the probabilistic approach discussed in this chapter, but this has become the default in recent years. In fact, any loss function that aims to reduce the distance between the model output and the training outputs will suffice and distance can be defined in any way that seems sensible. There are several well-known non-probabilistic machine learning models for classification, including support vector machines (Vapnik 1995; Cristianini & Shawe-Taylor 2000), which use *hinge loss*, and AdaBoost (Freund & Schapire 1995), which uses *exponential loss*.

Problems

Problem 5.1 Fill in the missing steps between lines two and three of equation 5.18.

Problem 5.2 Construct a loss function for making multivariate predictions $\mathbf{y}$ based on independent normal distributions as in section 5.4, but this time assume that we also want to estimate a separate variance σ_d^2 for each dimension. Assume these variances are constant, so the model is homoscedastic (*i.e.*, the variances do not vary with the data).

Problem 5.3 Construct a loss function for making multivariate predictions $\mathbf{y}$ based on independent normal distributions with different variances for each dimension. However, this time assume that this is a heteroscedastic model so that both the means μ_d and variances σ_d^2 change as a function of the data.

Problem 5.4 Show that the logistic sigmoid function $\text{sig}[z]$ maps $z = -\infty$ to 0, $z = 0$ to 0.5 and $z = \infty$ to 1 where:

$$\text{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{5.33}$$

Problem 5.5 The loss L for binary classification for a single training pair $\{\mathbf{x}, y\}$ is:

$$L = (1 - y)\log\left[1 - \text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]\right] + y\log\left[\text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]\right], \tag{5.34}$$

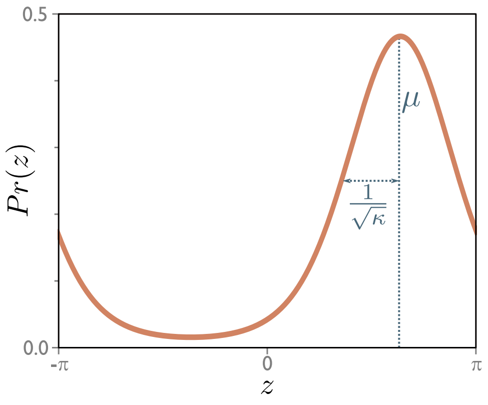


Figure 5.12 The von Mises distribution is defined over the circular domain $(-\pi, \pi]$. It has two parameters. The mean μ determines the position of the peak. The concentration $\kappa > 0$ acts like the inverse of the variance. Hence $1/\sqrt{\kappa}$ is roughly equivalent to the standard deviation in a normal distribution.

where sig[•] is defined in equation 5.33. Plot this loss as a function of the transformed network output $\text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]] \in [0, 1]$ (i) when the training label $y = 0$ and (ii) when $y = 1$.

Problem 5.6 Suppose we want to build a network that predicts the direction y in radians of the prevailing wind based on local measurements of barometric pressure $\mathbf{x}$. A suitable distribution over circular domains is the von Mises distribution (figure 5.12):

$$Pr(y|\mu, \kappa) = \frac{\exp[\kappa \cos[y - \mu]]}{2\pi \cdot \text{Bessel}_0[\kappa]}, \tag{5.35}$$

where μ is a measure of the mean direction and κ is a measure of the concentration (*i.e.*, the inverse of the variance). The term $\text{Bessel}_0[\kappa]$ is a modified Bessel function of order 0. Use the recipe from section 5.2 to develop a loss function for learning the parameter μ of a model $\text{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict the most likely wind direction. Your solution should treat the concentration κ as constant. How would you perform inference?

Problem 5.7 Extend the model from problem 5.6 to predict both the wind direction and the wind speed and define the associated loss function.

Problem 5.8 Sometimes, the predictions y for a given input $\mathbf{x}$ are naturally multimodal as in figure 5.13a; there is more than one valid prediction for a given input. In this case, we might use a weighted sum of normal distributions as the distribution over the output. This is known as a *mixture of Gaussians* model . For example, a mixture of two Gaussians has distribution parameters $\boldsymbol{\theta} = \{\lambda, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2\}$ is defined by:

$$Pr(y|\lambda, \mu_1, \mu_2, \sigma_1^2, \sigma_2^2) = \frac{\lambda}{\sqrt{2\pi\sigma_1^2}} \exp\left[\frac{-(y-\mu_1)^2}{2\sigma_1^2}\right] + \frac{1-\lambda}{\sqrt{2\pi\sigma_2^2}} \exp\left[\frac{-(y-\mu_2)^2}{2\sigma_2^2}\right], \tag{5.36}$$

where $\lambda \in [0, 1]$ controls the relative weight of the two normal distributions, which have means μ_1, μ_2, and variances σ_1^2, σ_2^2 respectively. This model can represent a distribution with two peaks (figure 5.13b) or a distribution with a single peak but a more complex shape (figure 5.13c).

Use the recipe from section 5.2 to construct a loss function for training a model $\mathbf{f}[x, \boldsymbol{\phi}]$ that takes input x, has parameters $\boldsymbol{\phi}$, and predicts a mixture of two Gaussians. The loss should be based on I training data pairs $\{x_i, y_i\}$. What possible problems do you foresee when we perform inference in this model?

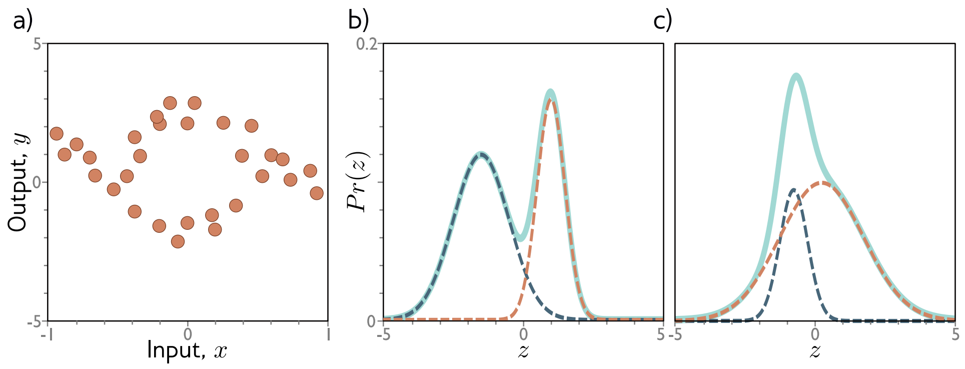


Figure 5.13 Multimodal data and mixture of Gaussians density. a) Example training data where for intermediate values of the input x, the corresponding output y follows one of two paths. For example, at $x = 0$, the output y might be roughly -2 or +3 but is unlikely to be between these values. b) The mixture of Gaussians is a probability model that is suited to this kind of data. As the name suggests, the model is a weighted sum (solid cyan curve) of two or more normal distributions with different means and variances (here two weighted distributions, dashed blue and orange curves). When the means are far apart, this forms a multimodal distribution. c) When the means are close, the mixture can model unimodal, but non-normal densities.

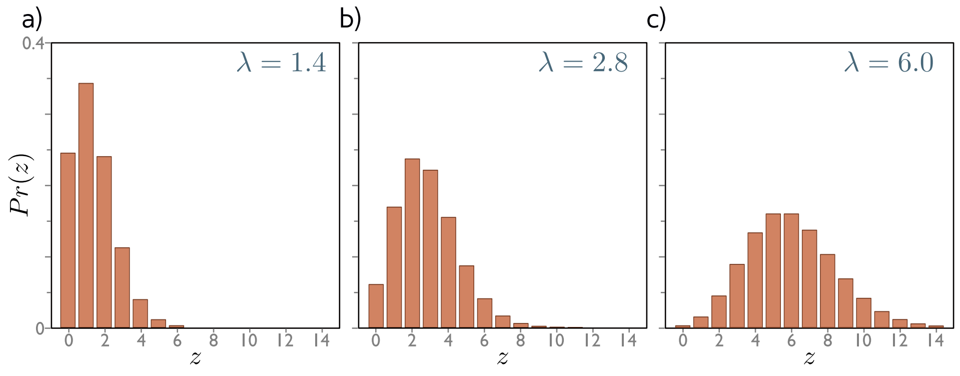


Figure 5.14 Poisson distribution. This discrete distribution is defined over non-negative integers $z \in \{0, 1, 2, \ldots\}$. It has a single parameter λ, which is known as the rate and is the mean of the distribution. a-c) Poisson distributions with rates of 1.4, 2.8, and 6.0 respectively.

Problem 5.9 Consider extending the model from problem 5.6 to predict the wind direction using a mixture of three von-Mises distributions. Write an expression for the likelihood $Pr(y|\boldsymbol{\theta})$ for this model. How many outputs will the network need to produce?

Problem 5.10 Consider building a model to predict the number of pedestrians $y \in \{0, 1, 2, \ldots\}$ that will pass a given point in the city in the next minute, based on data $\mathbf{x}$ that contains information about the time of day, the longitude and latitude, and the type of neighborhood. A suitable distribution for modeling counts is the Poisson distribution (figure 5.14). This has a single parameter $\lambda > 0$ called the *rate* that represents the mean of the distribution. The distribution has probability density function:

$$Pr(y = k) = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{5.37}$$

Use the recipe in section 5.2 to design a loss function for this model assuming that we have access to I training pairs $\{\mathbf{x}_i, y_i\}$.

Chapter 6

Fitting models

Chapters 3 and 4 described shallow and deep neural networks respectively. These represent families of piecewise linear functions, where the particular function is determined by the parameters. Chapter 5 introduced the loss – a single number that represents the mismatch between the network predictions and the ground truth targets for a training set.

The loss depends on the parameters of the network, and we now consider how to find parameter values that minimize this loss. This is known as *learning* the parameters of the network, or simply as *training* or *fitting* the model. The process is to choose initial parameter values and then iterate the following two steps: (i) compute the derivatives (gradients) of the loss with respect to the parameters, and (ii) adjust the parameters based on the gradients to decrease the loss. After many steps, we hope to reach the overall minimum of the loss function.

This chapter tackles the third of these steps; we consider algorithms that adjust the parameters to decrease the loss. Chapter 7 tackles the first and second problems; we discuss how to initialize the parameters and compute the gradients for neural networks.

6.1 Gradient descent

To fit a model, we need a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek parameters $\boldsymbol{\phi}$ for the model $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that map the inputs $\mathbf{x}_i$ to the outputs $\mathbf{y}_i$ as well as possible. To this end, we define a loss function $\mathrm{L}[\boldsymbol{\phi}]$ that returns a single number that quantifies the mismatch in this mapping. The goal of an *optimization algorithm* is to find parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss:

$$\hat{\boldsymbol{\phi}} = \operatorname{argmin}\left[\mathrm{L}[\boldsymbol{\phi}]\right]. \tag{6.1}$$

There are many families of optimization algorithms, but the standard methods for training neural networks are iterative. These algorithms initialize the parameters heuristically and then adjust them repeatedly in such a way that the loss decreases.

The simplest method in this class is *gradient descent*. This starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates two steps:

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{6.2}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{6.3}$$

where the positive scalar α determines the magnitude of the change.

The first step computes the gradient of the loss function at the current position. This determines the uphill direction on the loss function. The second step moves a small distance α *downhill* (hence the negative sign). The parameter α may be fixed (in which case, we call it a *learning rate*), or we may perform a *line search* where we test a number of different values of α to find the one that decreases the loss the most.

At the minimum of the loss function, the surface must be flat (or we could improve further by going downhill). Hence, the gradient will be zero, and the parameters will stop changing. In practice, we monitor the gradient magnitude, and when it becomes too small, we terminate the algorithm.

6.1.1 Linear regression example

Consider applying gradient descent to the 1D linear regression model from chapter 2. This only has two parameters and so is easy to visualize. The model $\mathrm{f}[x, \boldsymbol{\phi}]$ maps a scalar input x to a scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$ which represent the y-intercept and the slope:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{6.4}$$

Given a dataset $\{x_i, y_i\}$ containing I input/output pairs, we choose the least squares loss function:

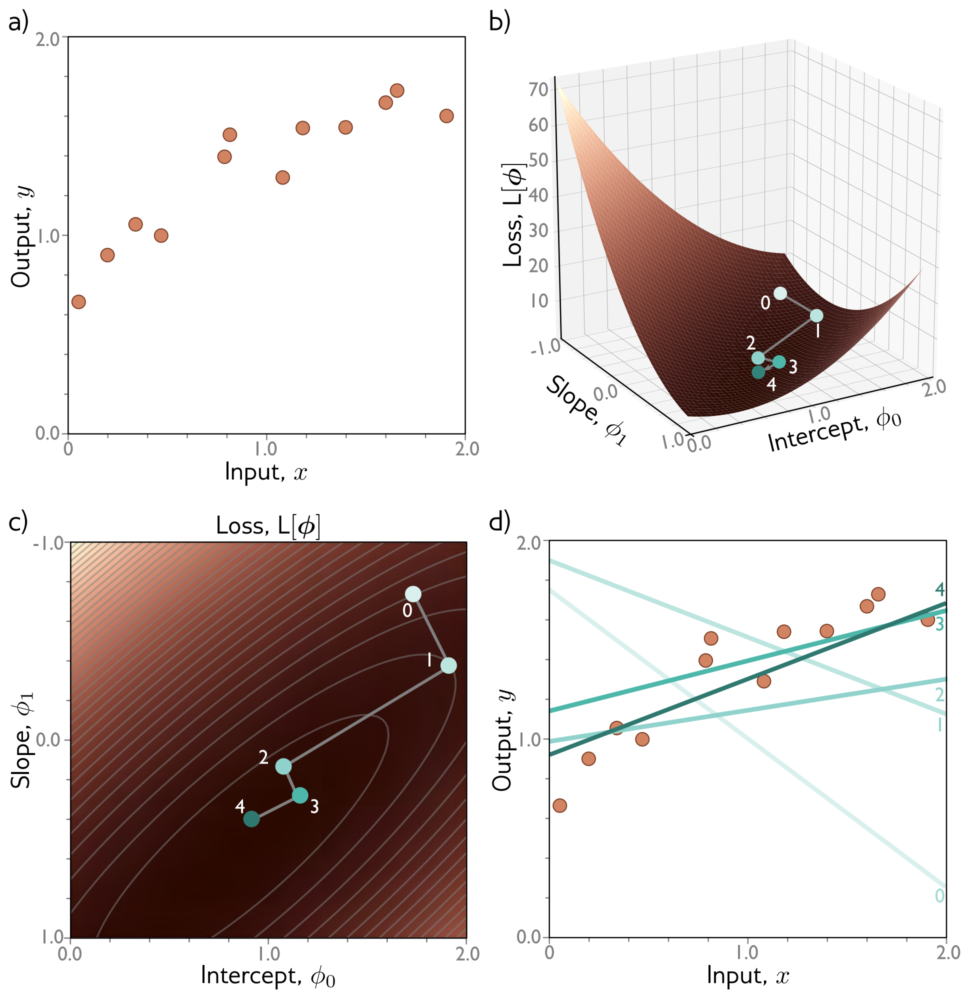


Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

$$\begin{aligned} L[\boldsymbol{\phi}] &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2 \\ &= \sum_{i=1}^{I} \mathrm{l}_i, \end{aligned} \tag{6.5}$$

where the term $l_i = (\phi_0 + \phi_1 x_i - y_i)^2$ is the individual contribution to the loss from the i^{th} training example.

The derivative of the loss function with respect to the parameters can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} l_i = \sum_{i=1}^{I} \frac{\partial l_i}{\partial \boldsymbol{\phi}}, \tag{6.6}$$

Problem 6.1

where these are given by:

$$\frac{\partial l_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial l_i}{\partial \phi_0} \\ \frac{\partial l_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}. \tag{6.7}$$

Figure 6.1 shows the progression of this algorithm as we iteratively compute the derivatives according to equations 6.6 and 6.7 and then update the parameters using the rule in equation 6.3. In this case, we have used a line search procedure to find the value of α that decreases the loss the most at each iteration.

6.1.2 Gabor model example

Problem 6.2

Loss functions for linear regression problems (figure 6.1c) always have a single well-defined global minimum. More formally, they are *convex* which means that no chord (line segment between two points on the surface) intersects the function. Convexity implies that wherever we initialize the parameters, we are bound to reach the minimum if we keep walking downhill; the training procedure really can't fail.

Unfortunately, loss functions for most non-linear models, including both shallow and deep networks, are *non-convex*. Visualization of neural network loss functions is challenging due to the number of parameters. Consequently, we'll first explore a simpler non-linear model with two parameters to gain insight into the properties of non-convex loss functions:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{8.0}\right). \tag{6.8}$$

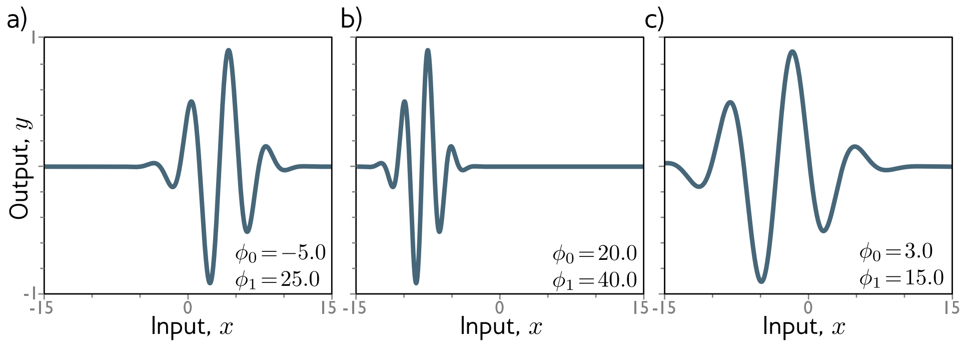


Figure 6.2 Gabor model. This non-linear model maps scalar input x to scalar output y and has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$. The model represents a sinusoidal function that decreases in amplitude with distance from the center. The parameter $\phi_0 \in \mathbb{R}$ determines the position of the center. As ϕ_0 increases, the function moves leftwards. The parameter $\phi_1 \in \mathbb{R}^+$ stretches or squeezes the function along the x-axis relative to the center. As ϕ_1 increases, the function narrows. a-c) Three examples of the model with different parameters.

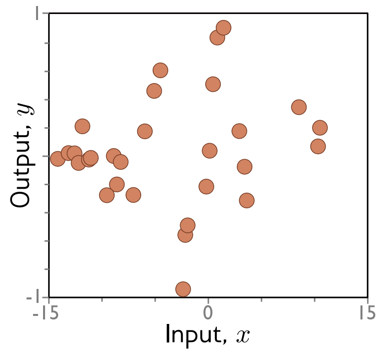


Figure 6.3 Training data for fitting Gabor model. The training dataset consists of 28 input/output examples $\{x_i, y_i\}$. These were created by uniformly sampling x_i in the range $[-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding noise.

This *Gabor model* maps scalar input x to scalar output y and consists of a sinusoidal component (creating an oscillatory function) multiplied by a negative exponential component (causing the amplitude to decrease as we move from the center). It has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where $\phi_0 \in \mathbb{R}$ determines the mean position of the function and $\phi_1 \in \mathbb{R}^+$ stretches or squeezes it along the x-axis (figure 6.2).

Problems 6.3-6.6

Consider a training set of I examples $\{x_i, y_i\}$ (figure 6.3). The least squares loss function for I training examples is defined as:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2. \tag{6.9}$$

Once more, the goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

6.1.3 Local minima and saddle points

Problem 6.4

Problem 6.7

The loss function associated with the Gabor model for this dataset is depicted in figure 6.4. There are numerous *local minima* (cyan circles). Here the gradient is zero and the loss increases if we move in any direction, but we are *not* at the overall minimum of the function. If we start in a random position and use gradient descent to go downhill, there is no guarantee that we will wind up at the global minimum (green point) and find the best parameters (figure 6.5a). It's equally or even more likely that the algorithm will terminate in one of the local minima. Furthermore, there is no way of knowing whether there is a better solution elsewhere.

In addition, the loss function contains *saddle points* (*e.g.*, the blue cross in figure 6.4). Here, the gradient is zero, but the function is increasing in some directions and decreasing in others. If the current position does not lie exactly on the saddle point, then gradient descent algorithms can escape by moving downhill. However, the surface near the saddle point is very flat and so it's hard to be sure that training has not converged.

6.2 Stochastic gradient descent

The Gabor model has two parameters and so we could find the global minimum by either (i) exhaustively searching the parameter space, or (ii) repeatedly starting gradient descent from different positions and choosing the result with the lowest loss. However, neural network models can have millions of parameters, and so for these models neither approach is practical. In short, using gradient descent to find the global optimum of a high-dimensional loss function is challenging. We can find *a* minimum, but there is no way to tell whether this is the global minimum or even a good one.

One of the main problems is that the final destination of a gradient descent algorithm is entirely determined by the starting point. *Stochastic gradient descent (SGD)* attempts to remedy this problem by adding some noise to the gradient at each step. The solution still moves downhill on average, but at any given iteration the direction chosen is not necessarily in the steepest downhill direction. Indeed, it might not be downhill at all. The SGD algorithm has the possibility of moving temporarily uphill and hence moving from one 'valley' of the loss function to another (figure 6.5b).

6.2.1 Batches and epochs

The mechanism for introducing randomness is simple. At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone. This subset is known as a *mini-batch* or *batch* for short. The update rule for the model parameters $\boldsymbol{\phi}_t$ at iteration t is hence:

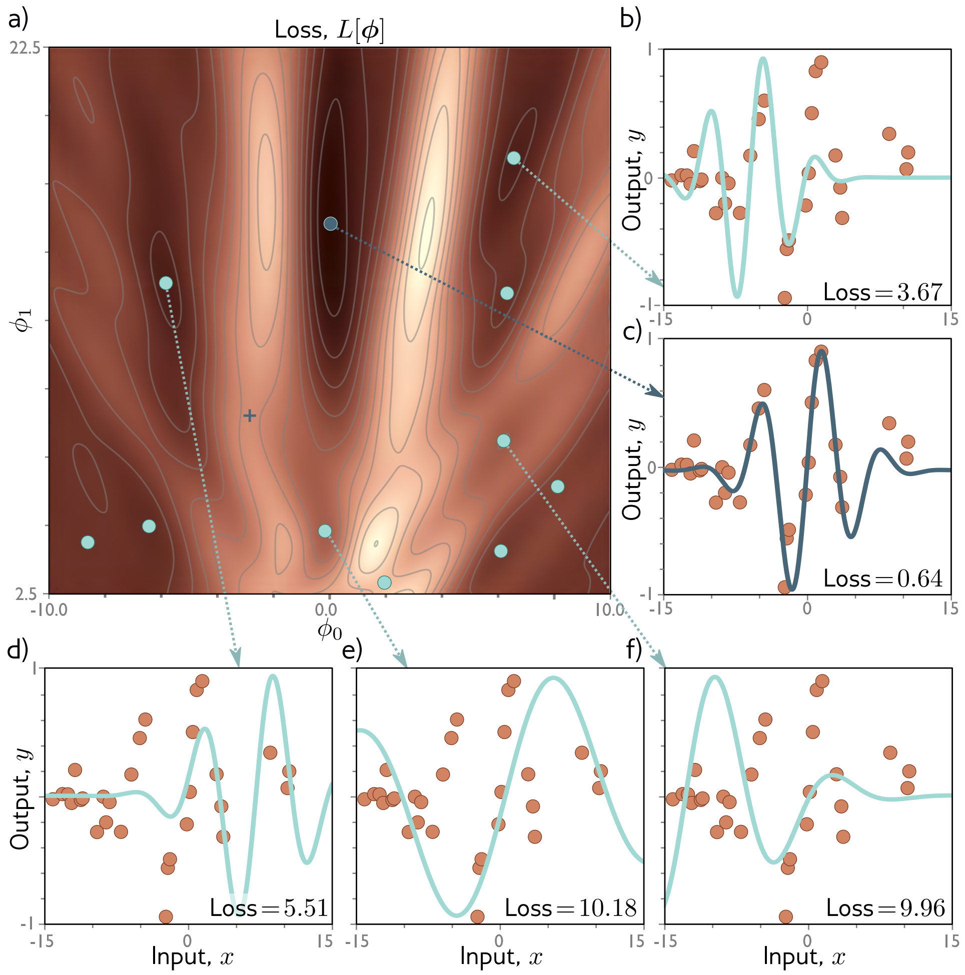


Figure 6.4 Loss function for Gabor model. a) The loss function is non-convex, with multiple local minima (cyan points) in addition to the global minimum (green point). It also contains saddle points where the gradient is locally zero, but the function is increasing in one direction and decreasing in the other. The green cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically. b-f) Models associated with the different minima. In each case, there is no small change that will decrease the loss. Panel (c) shows the best fitting model, which has a loss of 0.64.

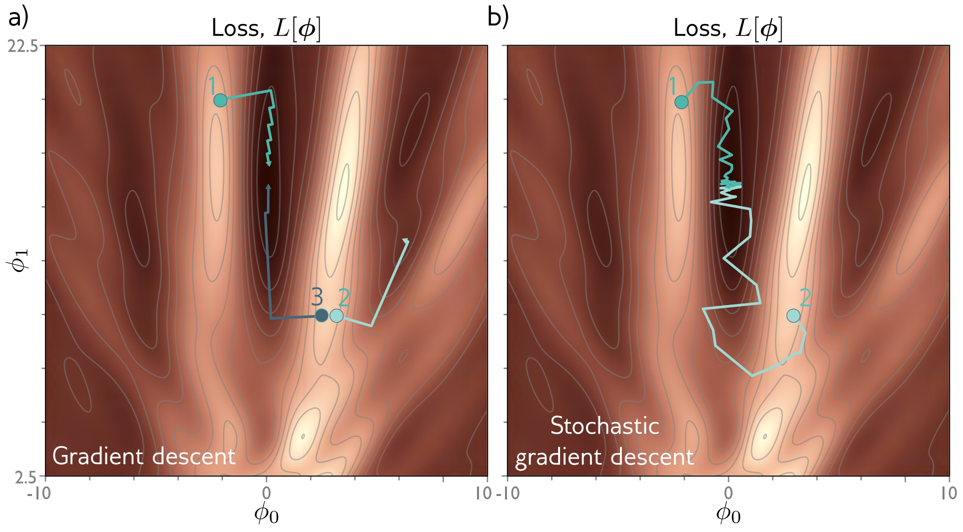


Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent. As long as the gradient descent algorithm is initialized in the right 'valley' of the loss function (*e.g.*, points 1 and 3), the parameter estimate will move steadily towards the global minimum. However, if it is initialized outside this valley (*e.g.*, point 2) then it will descend towards one of the local minima. b) Stochastic gradient descent adds noise to the optimization process and so it is possible to escape from starting points that are not in the right valley (*e.g.*, point 2) and still reach the global minimum.

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i\in\mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{6.10}$$

where $\mathcal{B}_t$ is a set containing the indices of the input/output pairs in the current batch and as before, l_i is the loss due to the i^{th} example. The term α is the learning rate and together with the gradient magnitude, this determines the distance moved at each iteration. The learning rate is chosen at the start of the procedure and does not depend on the local properties of the function.

The batches are usually drawn from the dataset without replacement. The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again. A single pass through all the data is referred to as an *epoch*. A batch may be as small as a single example, or as large the entire dataset. The latter case is referred to as *full-batch gradient descent* and is identical to regular (non-stochastic) gradient descent.

An alternative interpretation of SGD is that it computes the gradient of a different loss function at each iteration; the loss function depends on both the model and the training data, and so will be different for each randomly selected batch. In this view, SGD performs deterministic gradient descent on a constantly changing loss function (figure 6.6). However, despite this variability, the expected loss and expected gradients at any point remain the same as for gradient descent.

6.2.2 Properties of stochastic gradient descent

Stochastic gradient descent has several attractive features. First, although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration. Hence, the updates tend to be sensible even if they are not optimal. Second, because it draws training examples without replacement and iterates through the dataset, the training examples still all contribute equally. Third, it is less computationally expensive to compute the gradient from just a subset of the training data. Finally, there is some evidence that SGD finds parameters for neural networks that cause them to generalize well to new data in practice (see section 9.2).

Stochastic gradient descent does not necessarily 'converge' in the traditional sense. However, the hope is that when we are close to the global minimum, all the data points are described well by the model. Consequently, the gradient will be small whichever batch is chosen and the parameters will cease to change much. In practice, stochastic gradient descent is often applied with a *learning rate schedule*. The learning rate α starts at a relatively high value and is decreased by a constant factor every N epochs. The logic is that in the early stages of training, we want the algorithm to explore the parameter space, jumping from valley to valley to find a sensible region. In later stages, we are in roughly the right place and are more concerned with fine-tuning the parameters, and so we make smaller changes.

6.3 Momentum

A common modification to stochastic gradient descent is to add a *momentum* term. We update the parameters with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta\cdot\mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}}\\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha\cdot\mathbf{m}_{t+1},\end{aligned} \tag{6.11}$$

where $\mathbf{m}_t$ is the momentum term (which drives the update at iteration t), $\beta\in[0,1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.

The recursive formulation of the momentum calculation means that the gradient step is an infinite weighted sum of all the previous gradients, where the weights get smaller as we move back in time. The effective learning rate is increased if all these gradients are aligned over multiple iterations but decreased if the gradient

Problem 6.8

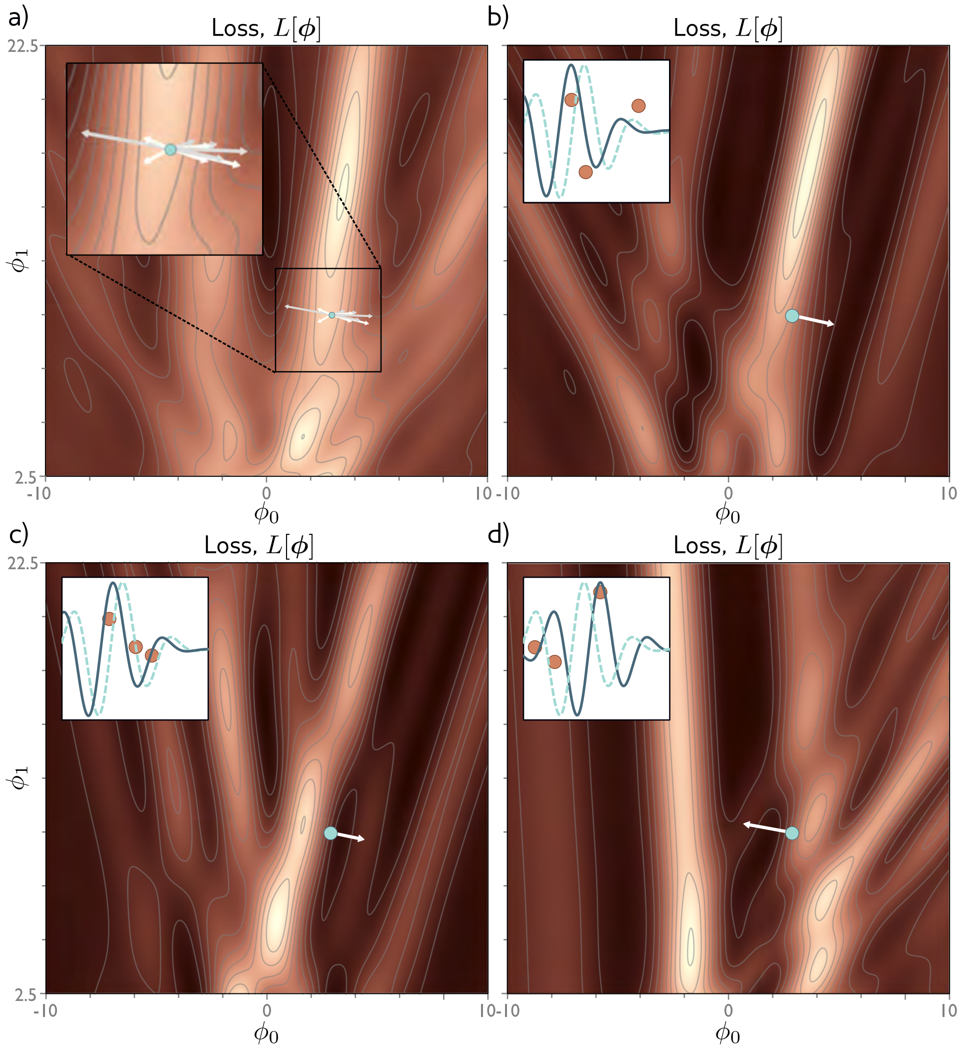


Figure 6.6 Alternative view of stochastic gradient descent for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. b) Loss function for one possible batch. The SGD algorithm moves in the downhill direction on this function for a distance that is determined by the learning rate and the local gradient magnitude. The current model (dashed function in inset) changes to fit the batch data better (solid function). c) A different choice of the batch would create a different loss function and hence result in a different parameter change. d) For this batch, the algorithm moves downhill with respect to the batch loss function but *uphill* with respect to the global loss function in panel (a). This is how SGD can escape local minima.

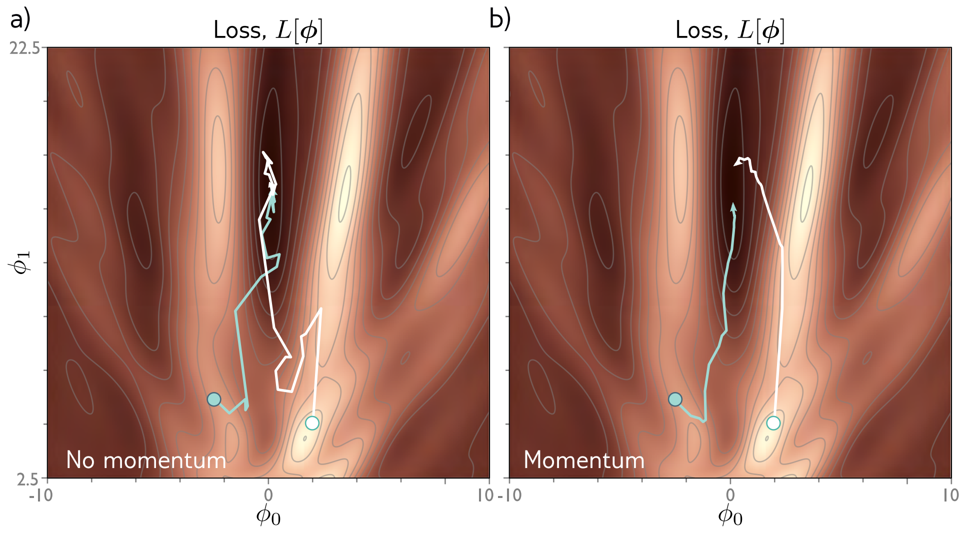


Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path towards the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

direction repeatedly changes as the terms in the sum cancel out. The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys (figure 6.7).

6.3.1 Nesterov accelerated momentum

The momentum term can be thought of as a coarse prediction of where the SGD algorithm will move next. Nesterov accelerated momentum (figure 6.8) computes the gradients at this predicted point rather than at the current point:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta\cdot\mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial l_i[\boldsymbol{\phi}_t-\alpha\cdot\mathbf{m}_t]}{\partial\boldsymbol{\phi}}\\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha\cdot\mathbf{m}_{t+1},\end{aligned} \tag{6.12}$$

where now the gradients are evaluated at $\boldsymbol{\phi}_t - \alpha\cdot\mathbf{m}_t$. One way to think about this is that the gradient term now corrects the path provided by momentum alone.

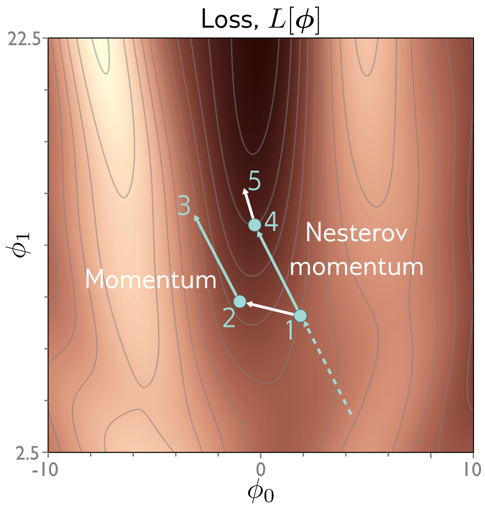


Figure 6.8 Nesterov accelerated momentum. The optimization trajectory has just traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1 and moves some distance in this direction to point 2 and then adds the momentum term from the previous iteration, (*i.e.*, in the same direction as the dashed line) arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4), and then measures the gradient and applies an update to arrive at point 5.

6.4 Adam

Gradient descent with a fixed step size has the following undesirable property. It makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious), and small adjustments to parameters associated with small gradients (where perhaps we should explore further). This means that when the gradient of the loss surface is much steeper in one direction than another, it is very difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable (figures 6.9a-b).

A very simple approach is to normalize the gradients so that we move a fixed distance governed by the learning rate in each direction. To do this, we first measure the gradient $\mathbf{m}_{t+1}$ and the squared gradient $\mathbf{v}_{t+1}$:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\\ \mathbf{v}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}^2.\end{aligned} \tag{6.13}$$

Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \quad\leftarrow\quad \boldsymbol{\phi}_t - \alpha\cdot\frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}}+\epsilon}, \tag{6.14}$$

where the square root and division are both pointwise, α is the learning rate, and ϵ is a small constant that prevents division by zero when the gradient magnitude is zero. The term $\mathbf{v}_{t+1}$ is the squared gradient, and the positive root of this is used to normalize the gradient itself so all that remains is the sign in each coordinate direction. The result is that the algorithm moves a fixed distance α along each

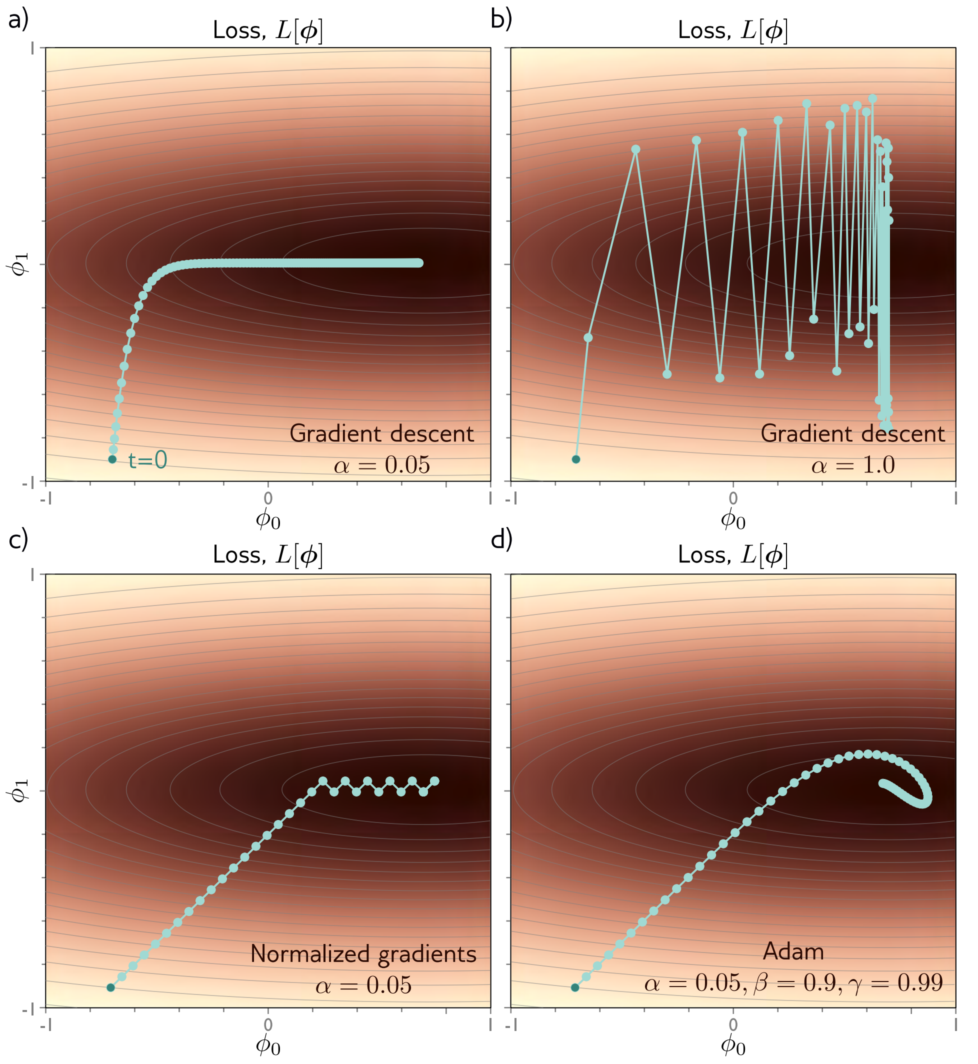


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction, but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a very long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, then it overshoots in the vertical direction and becomes unstable. c) A very simple approach is just to move a fixed distance along each axis at each step in such a way that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude, retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term which together create a smoother path.

coordinate, where the direction is determined by whichever way is downhill (figure 6.9c). This simple algorithm makes good progress in both directions but will not converge unless it happens to land exactly at the minimum. Instead, it will bounce back and forth around the minimum.

Adaptive moment estimation or *Adam* takes this idea and adds momentum to both the estimate of the gradient and the squared gradient:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta\cdot\mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}}\\ \mathbf{v}_{t+1} &\leftarrow \gamma\cdot\mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}}\right)^2,\end{aligned}\tag{6.15}$$

where β and γ are the momentum coefficients for the two statistics.

Using momentum is equivalent to taking a weighted average over the history of each of these statistics. At the start of the procedure, all the previous measurements are effectively zero and so this results in unrealistically small estimates. Consequently, we modify these statistics using the rule:

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}}\\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned}\tag{6.16}$$

Since β and γ are in the range $[0,1)$, the terms with exponents $t+1$ become smaller with each time-step, and so the denominator becomes closer to one and the modification has a diminishing effect.

Finally, we update the parameters as before, but with the modified terms:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha\cdot\frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}}+\epsilon}.\tag{6.17}$$

The result is an algorithm that can converge to the overall minimum and makes good progress in every direction in parameter space. Note that Adam is usually used in a stochastic setting where the gradients and their squares are computed from mini-batches:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta\cdot\mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}}\\ \mathbf{v}_{t+1} &\leftarrow \gamma\cdot\mathbf{v}_t + (1-\gamma)\sum_{i\in\mathcal{B}_t}\left(\frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}}\right)^2,\end{aligned}\tag{6.18}$$

and so the trajectory will be noisy in practice.

As we shall see chapter 7, the gradient magnitudes of neural network parameters can depend on their depth in the network. Adam helps compensate for this tendency and balances out changes across the different layers. In practice, Adam also has the advantage that it is less sensitive to the initial learning rate, because it avoids situations like those in figures 6.9a-b, and so it doesn't need complex learning rate schedules.

6.5 Training algorithm hyperparameters

The choice of learning algorithm, batch size, the learning rate schedule, and the momentum coefficients are all considered *hyperparameters* of the training algorithm; these directly affect the quality of the final model but are distinct from the model parameters. Choosing these is more art than science and it's common to train many models with different hyperparameters and choose the best one. This is known as *hyperparameter tuning*. We return to this issue in chapter 8.

6.6 Summary

In this chapter, we've discussed model training. This problem is framed as finding the parameters $\boldsymbol{\phi}$ that correspond to the minimum of a loss function $L[\boldsymbol{\phi}]$. In the gradient descent method, we measure the gradient of the loss function for the current parameters (*i.e.*, how the loss changes when we make a small change to the parameters). Then we move in the direction that decreases the loss fastest. This is repeated until we can improve no further.

Unfortunately, for non-linear functions, the loss function may have both local minima (where the gradient descent algorithm will get trapped) and saddle points (where the gradient descent algorithm may appear to have converged). We introduced stochastic gradient descent, which helps prevent these problems. At each iteration, we use a different random subset of the data (a batch) to compute the gradient. This adds noise to the process and helps prevent the algorithm from getting trapped in a sub-optimal region of parameter space. Each iteration is also computationally cheaper. We discussed adding a momentum term to make convergence more efficient, and finally we introduced the Adam algorithm.

The ideas in the chapter are applicable to optimizing *any* model. In the next chapter, we tackle two aspects of training that are specific to neural networks. First, we address how to compute the gradients of the loss with respect to neural network parameters. This is accomplished using the famous backpropagation algorithm. Second, we discuss how to initialize the network parameters before the optimization begins. Without careful initialization, the gradients used by the optimization can become extremely large or extremely small, and this can hinder the training process.

Notes

Optimization algorithms Optimization algorithms are used extensively throughout science and engineering. In the wider literature, it is more typical to talk about the *objective function* rather than the loss function or cost function. Gradient descent was invented by Cauchy (1847) and stochastic gradient descent dates back to at least Robbins & Monro (1951). A modern compromise between the two is stochastic variance-reduced descent (Johnson & Zhang 2013) in which the full gradient is computed periodically with stochastic updates interspersed. Reviews of optimization algorithms for neural networks can be found in Ruder (2016), Bottou *et al.* (2018), and Sun (2020). Bottou (2012) discuss best practice for SGD, including shuffling without replacement.

Convexity, minima, and saddle points: A function is convex if no chord (line segment between two points on the surface) intersects the function. This can be tested algebraically by considering the *Hessian matrix* (the matrix of second derivatives):

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_N} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} & \cdots & \frac{\partial^2 L}{\partial \phi_1 \partial \phi_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial \phi_N \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_N \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_N \partial \phi_N} \end{bmatrix}. \tag{6.19}$$

Appendix C.1.4 Eigenvalues

If the Hessian matrix is positive definite (has positive eigenvalues) for all possible parameter values, then the function is convex; the loss function will look like a smooth bowl (as in figure 6.1c) and so training is relatively easy. There will be a single global minimum and no local minima or saddle points.

For any loss function, the eigenvalues of the Hessian matrix at places where the gradient is zero allow us to classify this position as (i) a minimum (the eigenvalues are all positive), (ii) a maximum (the eigenvalues are all negative) or (iii) a saddle point (positive eigenvalues are associated with directions in which we are at a minimum and negative ones with directions where we are at a maximum).

Line search: Gradient descent with a fixed step size is inefficient. In this case, the distance moved depends entirely on the magnitude of the gradient. It follows that we move a long distance when the function is changing fast (where perhaps we should be more cautious), but a short distance when the function is changing slowly (where perhaps we should explore further). For this reason, gradient descent methods are usually combined with a line search procedure in which we sample the function along the desired direction to try to find the optimal step size. One such approach is bracketing (figure 6.10). Another problem with gradient descent is that it tends to lead to inefficient oscillatory behavior when descending valleys (*e.g.*, path 1 in figure 6.5a).

Beyond gradient descent: Numerous algorithms have been developed that remedy the problems of gradient descent. Most notable is the Newton method which takes the curvature of the surface into account using the inverse of the Hessian matrix; if the gradient of the function is changing quickly, then it applies a more cautious update. This method eliminates the need for line search and does not suffer from oscillatory behavior. However, it has its own problems; in its simplest form, it moves towards the nearest extremum, but this may be a maximum if we are closer to the top of a hill than we are to the bottom of a valley. Moreover, computing the inverse Hessian is intractable when the number of parameters is large as in neural network models.

Problem 6.9

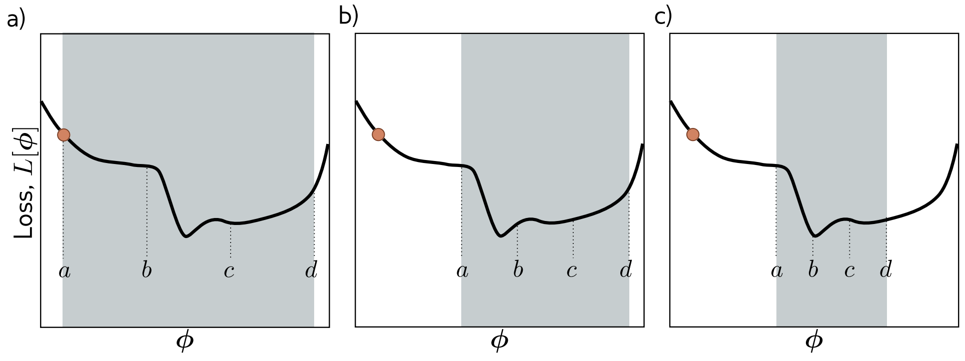


Figure 6.10 Line search using bracketing approach. a) The current solution is at position a (orange point) and we wish to search the region $[a, d]$ (gray shaded area). We define two points b, c interior to the search region and evaluate the loss function at these points. Here $L[b] > L[c]$ and so we eliminate the range $[a, b]$. b) We now repeat this procedure in the refined search region and find that $L[b] < L[c]$ and so we eliminate the range $[c, d]$. c) We repeat this process until this minimum is closely bracketed.

Exhaustive search: All the algorithms discussed in this chapter are iterative; A completely different approach is to quantize the network parameters and to exhaustively search the resulting discretized parameter space using SAT solvers (Mezard & Mora 2008). This approach has the potential to find the global minimum and provide a guarantee that there is no lower loss elsewhere but is only practical for very small models.

Momentum: The idea of using momentum to speed up optimization algorithms dates to Polyak (1964). Goh (2017) presents an in-depth discussion of the properties of momentum. The Nesterov accelerated gradient method was introduced in Nesterov (1983). The Nesterov momentum approach was first applied in the context of batch gradient descent by Sutskever *et al.* (2013).

Adaptive training algorithms: AdaGrad (Duchi *et al.* 2011) is an optimization algorithm that addresses the possibility that some parameters may have to move further than others by assigning a different learning rate to each parameter. AdaGrad uses the cumulative squared gradient for each parameter to attenuate its learning rate. This has the disadvantage that the learning rates decrease over time and learning can halt before the minimum is found. Both RMSProp (Hinton *et al.* 2012a) and AdaDelta (Zeiler 2012) modified this algorithm to help prevent this problem by updating the squared gradient term recursively.

By far the most widely used adaptive training algorithm is adaptive moment optimization or Adam (Kingma & Ba 2014). This combines the ideas of momentum (in which the gradient vector is averaged over time) and AdaGrad, AdaDelta, and RMSProp (in which a smoothed squared gradient term is used to modify the learning rate for each parameter). The original paper on the Adam algorithm provided a convergence proof for convex loss functions, but a counterexample was identified by Reddi *et al.* (2018) who developed a modification of Adam called AMSGrad, which does converge. Of course, in deep learning the loss functions are non-convex and Zaheer *et al.* (2018) subsequently developed an adaptive algorithm called YOGI, and proved that it converges in this scenario. Regardless of these theoretical objections, the original Adam algorithm works well in practice and is widely used, not least because it works well over a broad range of hyperparameters and makes rapid initial progress

One potential problem with adaptive training algorithms is that the learning rates are based on accumulated statistics of the observed gradients. At the start of training, when there are few samples, these statistics may be very noisy. This can be remedied by *learning rate warm-up* (Goyal *et al.* 2018), in which the learning rates are gradually increased over the first few thousand iterations. An alternative solution is rectified Adam (Liu *et al.* 2021a) which gradually changes the momentum term over time in a way that helps avoid high variance. Dozat (2016) incorporated Nesterov momentum into the Adam algorithm.

SGD vs. Adam: There has been a lively discussion about the relative merits of SGD and Adam. Wilson *et al.* (2018) provided evidence that SGD with momentum can find lower minima than Adam that generalize better over a variety of deep learning tasks. However, this is strange since SGD is a special case of Adam (when $\epsilon = \gamma = 0$) once the modification term (equation 6.16) becomes one, which happens quickly. It is hence more likely that SGD outperforms Adam *when we use Adam's default hyperparameters.* Loshchilov & Hutter (2017) proposed a variation of Adam called AdamW, which substantially improves the performance of Adam in the presence of L2 regularization (see section 9.1). Choi *et al.* (2019) provide evidence that if we search for the best Adam hyperparameters, then it performs just as well as SGD and converges faster. Keskar & Socher (2017) proposed a method called SWATS that starts using Adam (to make rapid initial progress) and then switches to SGD (to get better final generalization performance).

Problems

Problem 6.1 Show that the derivatives of the least squares loss function in equation 6.5 are given by the expressions in equation 6.7.

Problem 6.2 A surface is convex if the eigenvalues of the Hessian $H[\boldsymbol{\phi}]$ are positive everywhere. In this case, the surface has a unique minimum and optimization is easy. Find an algebraic expression for the Hessian matrix,

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} \end{bmatrix}, \tag{6.20}$$

Appendix C.1.4 Eigenvalues
Appendix C.1.1 Trace
Appendix C.1.2 Determinant

for the linear regression model (equation 6.5). Prove that this function is convex by showing that the eigenvalues are always positive. This can be done by showing that both the trace and the determinant of the matrix are positive.

Problem 6.3 Compute the derivatives of the least squares loss $L[\boldsymbol{\phi}]$ with respect to the parameters ϕ_0 and ϕ_1 for the Gabor model (equation 6.8).

Problem 6.4 Which of the three functions in figure 6.11 is convex? Justify your answer. Characterize each of the points 1-7 as (i) local minima, (ii) global minima, or (iii) other.

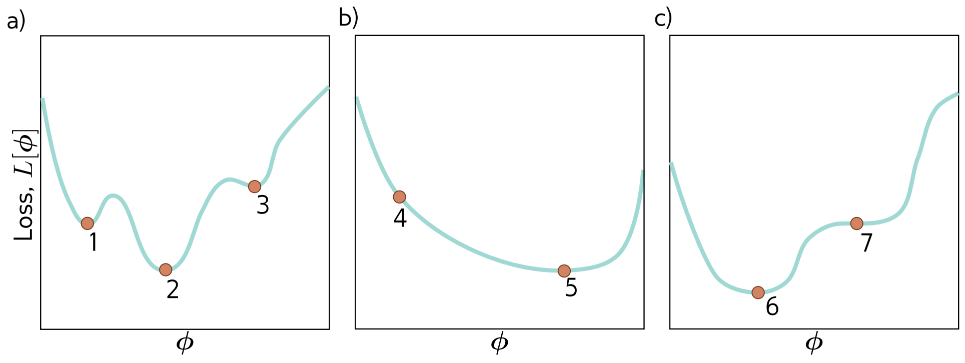


Figure 6.11 Three one dimensional loss functions for problem 6.4.

Problem 6.5 The logistic regression model uses a linear function to predict which of two classes $y \in \{0, 1\}$ an input $\mathbf{x}$ belongs to. For a 1D input and a 1D output, it has two parameters ϕ_0 and ϕ_1, and is defined by:

$$Pr(y = 1|x) = \text{sig}[\phi_0 + \phi_1 x], \tag{6.21}$$

where sig[•] is the logistic sigmoid function:

$$\text{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{6.22}$$

(i) Plot y against x for this model for different values of ϕ_0 and ϕ_1 and explain the qualitative meaning of each parameter. (ii) What is a suitable loss function for this model? (iii) Compute the derivatives of this loss function with respect to the parameters. (iv) Generate ten data points from a normal distribution with mean minus one and standard deviation one and assign the label $y = 0$ to them. Generate another ten data points from a normal distribution with mean one and standard deviation one and assign the label $y = 1$ to these. Plot the loss as a heatmap in terms of the two parameters ϕ_0 and ϕ_1. (v) Is this loss function convex? How could you prove this?

Problem 6.6 Compute the derivatives of the least squares loss with respect to the ten parameters of the simple neural network model introduced in equation 3.1:

$$y = \phi_0 + \phi_1 \text{a}[\theta_{10} + \theta_{11}x] + \phi_2 \text{a}[\theta_{20} + \theta_{21}x] + \phi_3 \text{a}[\theta_{30} + \theta_{31}x]. \tag{6.23}$$

Think carefully about what the derivative of the ReLU function $a[\bullet]$ will be.

Problem 6.7 The gradient descent trajectory for path 1 in figure 6.5a oscillates back and forth inefficiently as it moves down the valley towards the minimum. It's also notable that it turns at right angles to the previous direction at each step. Provide a qualitative explanation for these phenomena. Propose a solution that might help prevent this behavior.

Problem 6.8 Show that the momentum term $\mathbf{m}_t$ (equation 6.11) is an infinite weighted sum of the gradients at the previous iterations and derive an expression for the coefficients (weights) of that sum.

Problem 6.9 What dimensions will the Hessian have if the model has one million parameters?

Chapter 7

Gradients and initialization

Chapter 6 introduced iterative optimization algorithms. These are general purpose methods for finding the minimum of a function. In the context of neural networks, they find parameters that minimize the loss so that the model accurately predicts the training outputs from the inputs. The basic approach is to choose initial parameters randomly, and then make a series of small changes that decrease the loss on average. Each change is based on the gradient of the loss with respect to the parameters at the current position.

This chapter discusses two issues that are specific to neural networks. First, we consider how to calculate the gradients efficiently. This is a serious challenge since the largest models at the time of writing have $\sim 10^{12}$ parameters, and the gradient needs to be computed for every parameter and at every iteration of the training algorithm. Second, we consider how to initialize the parameters. If this is not done carefully, the initial losses and their gradients can be extremely large or extremely small. In either case, this impedes the training process.

7.1 Problem definitions

Let's consider a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with multivariate input $\mathbf{x}$, parameters $\boldsymbol{\phi}$, and three hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \\
\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3,
\end{aligned} \tag{7.1}$$

where the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of the input. The vectors $\boldsymbol{\beta}_k$ contain the bias terms and the matrices $\boldsymbol{\Omega}_k$ contain the weights (figure 7.1). Collectively these comprise the model parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_0, \boldsymbol{\Omega}_0, \boldsymbol{\beta}_1, \boldsymbol{\Omega}_1, \boldsymbol{\beta}_2, \boldsymbol{\Omega}_2, \beta_3, \boldsymbol{\Omega}_3\}$.

We'll assume that we know the individual loss term $\text{l}[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], y_i]$, which returns the negative log-likelihood of the ground truth label y_i given the model prediction $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for training input $\mathbf{x}_i$. For example, it might be the multi-class classification loss (equation 5.27). The total loss is the sum of these terms over the training data:

$$\begin{aligned} L[\boldsymbol{\phi}] &= \sum_{i=1}^{I} \text{l}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}], y_i] \\ &= \sum_{i=1}^{I} l_i, \end{aligned} \tag{7.2}$$

where l_i is shorthand for the individual loss term for the i^{th} training example.

The most commonly used optimization algorithm for training neural networks is stochastic gradient descent (SGD), which updates the parameters as:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{7.3}$$

where α is the learning rate and $\mathcal{B}_t$ contains the batch indices at iteration t. To compute this update, we need to calculate terms of the form:

$$\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} \quad \text{and} \quad \frac{\partial l_i}{\partial \boldsymbol{\beta}_k}, \tag{7.4}$$

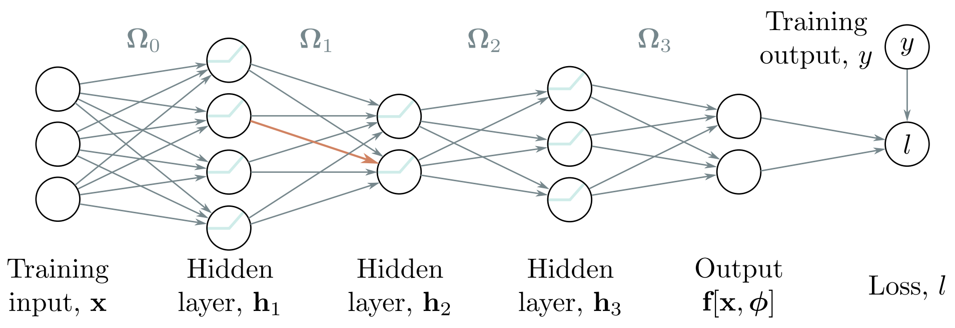
Problem 7.1

for the weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$ at every layer $k \in \{0, 1, \ldots, K\}$ and for each index i in the batch. The first part of this chapter describes the *backpropagation algorithm*, the purpose of which is to compute these derivatives efficiently.

In the second part of the chapter, we consider how to initialize the network parameters before we commence training. We describe methods to choose the initial weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$ so that training is stable.

7.2 Computing derivatives

The derivatives of the loss tell us how the loss changes when we make a small change to the parameters. Optimization algorithms exploit this information to manipulate the parameters so that the loss becomes smaller. The *backpropagation algorithm* computes these derivatives. The mathematical details are somewhat involved, and so before describing them, we first make two observations that provide some intuition.



Ω_0
Ω_1
Ω_2
Ω_3
Training output, y
y
l
Training input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$
Loss, l

Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss l with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the orange weight multiplies the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the orange weight will double too. It follows that to compute the derivatives of the weights, we need to know the activations at the hidden layers. Computing and storing these is referred to as the *forward pass*, since it involves running the network equations sequentially for the data example.

Observation 1: Each weight (element of $\boldsymbol{\Omega}_k$) multiplies the activation at a source hidden unit and adds the result to a destination hidden unit in the next layer. It follows that the effect of any small change to the weight is amplified or attenuated by the activation at the source hidden unit. Hence, we will have to compute and store the activations of all of the hidden units in order to compute the gradients. This is known as the *forward pass* (figure 7.1).

Observation 2: A small change in a bias or weight causes a ripple effect of changes through the subsequent network. The change modifies the value of its destination hidden unit. This in turn changes the values of the hidden units in the subsequent layer, which will change the hidden units in the layer after that and so on, until a change is made to the model output and finally the loss.

Hence, to know how changing a parameter modifies the loss, we also need to know how changes to every subsequent hidden layer will in turn modify their successor. These same quantities are needed when we consider other parameters in the same layer or parameters in earlier layers. It follows that we can calculate them once and re-use them. For example, consider computing the effect of a small change in weights that feed into hidden layers $\mathbf{h}_3$, $\mathbf{h}_2$, and $\mathbf{h}_1$ respectively:

- To calculate how a small change in a weight or bias feeding into hidden layer

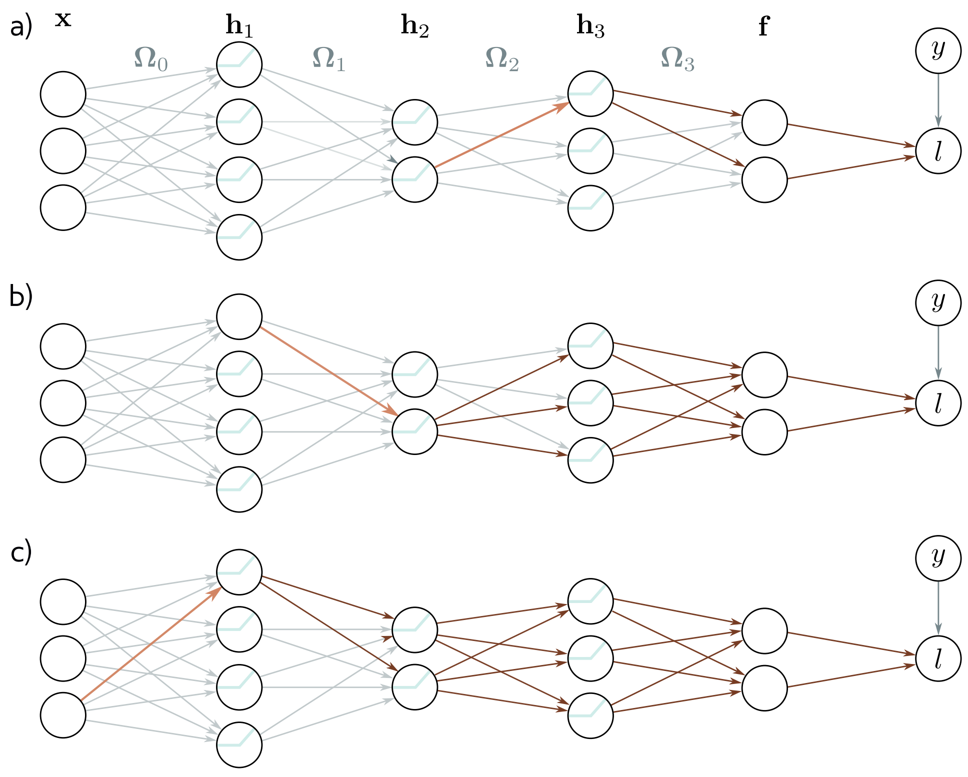


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (orange arrow) changes the loss, we need to know how the neuron in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (brown arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (orange arrow) changes the loss, we need to know (i) how the neuron in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (brown arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (orange arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (brown arrows). The backward pass computes derivatives at the end of the network first and works backward to exploit the inherent redundancy of these computations.

$\mathbf{h}_3$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_3$ changes the model output $\mathbf{f}$, and (ii) how a change in model output changes the loss l (figure 7.2a).

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_2$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_2$ affects $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes the model output, and (iii) how this output changes the loss (figure 7.2b).
- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_1$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_1$ affects layer $\mathbf{h}_2$, (ii) how a change in layer $\mathbf{h}_2$ affects layer $\mathbf{h}_3$, (iii) how layer $\mathbf{h}_3$ changes the model output, and (iv) how the model output changes the loss (figure 7.2c).

As we move backward through the network, we see that most of the terms that we need have already been computed in the previous step. and so we can re-use this computation. Proceeding backward through the network in this way to compute the derivatives is known as the *backward pass*.

7.2.1 Backpropagation

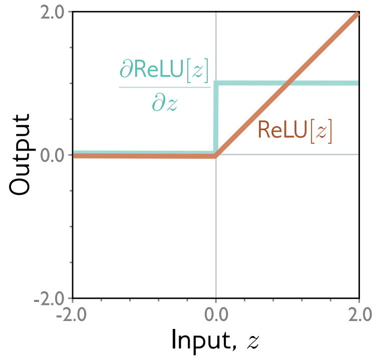
The backpropagation algorithm follows from these two intuitions and consists of (i) a forward pass, in which we compute and store the values at all of the hidden units and the network output, and (ii) a backward pass, in which we calculate the derivatives of each parameter, starting at the end of the network, and reusing the previous computation as we move towards the start.

Forward pass: We can think of the network in equation 7.1 as a series of sequential calculations:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_1 &= \mathbf{a}[\mathbf{f}_0] \\
\mathbf{f}_1 &= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1 \\
\mathbf{h}_2 &= \mathbf{a}[\mathbf{f}_1] \\
\mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2 \\
\mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] \\
\mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3 \\
l_i &= \mathrm{l}[\mathbf{f}_3, y_i],
\end{aligned} \tag{7.5}$$

where $\mathbf{f}_{k-1}$ represents the pre-activations at the k^{th} hidden layer (*i.e.*, the values before the ReLU function $\mathbf{a}[\bullet]$) and $\mathbf{h}_k$ contains the activations at the k^{th} hidden

Figure 7.3 Derivative of rectified linear unit. The rectified unit (orange curve) returns zero when the input is less than zero and returns the input otherwise. Its derivative (cyan curve) returns zero when the input is less than zero (since the slope here is zero) and one when the input is greater than zero (since the slope here is one).



layer (*i.e.*, after the ReLU function). In the forward pass, we simply work through these calculations and store all the intermediate quantities.

Matrix calculus Appendix C.2

Backward pass #1: Now let's consider how the loss changes when we modify the pre-activations $\mathbf{f}_0, \mathbf{f}_1, \mathbf{f}_2$. Applying the chain rule, the expression for the derivative of the loss l_i with respect to $\mathbf{f}_2$ is:

$$\frac{\partial l_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial l_i}{\partial \mathbf{f}_3}. \tag{7.6}$$

The three terms on the right-hand side have sizes $D_3 \times D_3, D_3 \times D_f$, and $D_f \times 1$ respectively, where D_3 is the number of hidden units in the third layer and D_f is the dimensionality of the model output $\mathbf{f}_3$.

Similarly, we can compute how the loss changes when we change $\mathbf{f}_1$ and $\mathbf{f}_0$:

$$\frac{\partial l_i}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial l_i}{\partial \mathbf{f}_3}\right) \tag{7.7}$$

$$\frac{\partial l_i}{\partial \mathbf{f}_0} = \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_0}\frac{\partial \mathbf{f}_1}{\partial \mathbf{h}_1}\left(\frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial l_i}{\partial \mathbf{f}_3}\right). \tag{7.8}$$

Note that in each case, the term in brackets was computed in the previous step. By working backward through the network, we can re-use the previous computations.

Problems 7.2-7.3

Moreover, the terms themselves are extremely simple. Working backward through the right-hand side of equation 7.6, we have:

- The derivative $\partial l_i/\partial \mathbf{f}_3$ of the loss l_i with respect to the network output $\mathbf{f}_3$ will depend on the loss function but usually has a simple form.

Problem 7.4

- The derivative $\partial \mathbf{f}_3/\partial \mathbf{h}_3$ of the network output with respect to the last hidden layer $\mathbf{h}_3$ is:

$$\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} = \frac{\partial}{\partial \mathbf{h}_3}\left(\boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3\right) = \boldsymbol{\Omega}_3^T. \tag{7.9}$$

If you are not familiar with matrix calculus, then this result is not obvious. It is explored in problem 7.4.

- The derivative $\partial \mathbf{h}_3/\partial \mathbf{f}_2$ of the output $\mathbf{h}_3$ of the activation function with respect to its input $\mathbf{f}_2$ will depend on the activation function. For ReLU functions, this is a diagonal matrix where the diagonal terms are zero everywhere $\mathbf{f}_2$ is less than zero and one everywhere it is greater (figure 7.3). Rather than multiply by this matrix, we extract the diagonal terms as a vector $\mathbb{I}[\mathbf{f}_2 > 0]$ and pointwise multiply, which is more efficient.

Problems 7.5-7.6

The terms on the right-hand side of equations 7.7 and 7.8 have similar forms. As we progress back through the network, we alternately (i) multiply by the transpose of the weight matrices $\boldsymbol{\Omega}_k^T$ and (ii) threshold based on the inputs $\mathbf{f}_{k-1}$ to the hidden layer. These inputs were stored during the forward pass.

Backward pass #2: Now that we know how to compute the derivatives $\partial l_i/\partial \mathbf{f}_k$, we can focus on calculating the derivatives of the loss with respect to the weights and biases. To calculate the derivatives of the loss with respect to the biases $\boldsymbol{\beta}_k$ we again use the chain rule:

$$\begin{aligned}\frac{\partial l_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\beta}_k}\frac{\partial l_i}{\partial \mathbf{f}_k}\\ &= \frac{\partial}{\partial \boldsymbol{\beta}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial l_i}{\partial \mathbf{f}_k}\\ &= \frac{\partial l_i}{\partial \mathbf{f}_k},\end{aligned} \tag{7.10}$$

which we already calculated in equations 7.6 and 7.7.

Similarly, the derivative for the weights vector $\boldsymbol{\Omega}_k$, is given by:

$$\begin{aligned}\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\Omega}_k}\frac{\partial l_i}{\partial \mathbf{f}_k}\\ &= \frac{\partial}{\partial \boldsymbol{\Omega}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial l_i}{\partial \mathbf{f}_k}\\ &= \frac{\partial l_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T.\end{aligned} \tag{7.11}$$

Again, the progression from line two to line three is not obvious and is explored in problem 7.7. However, the result makes sense. The final line is a matrix of the same size as $\boldsymbol{\Omega}_k$. It depends linearly on $\mathbf{h}_k$, which was multiplied by $\boldsymbol{\Omega}_k$ in the original expression. This is also consistent with the initial intuition that the derivative of the weights in $\boldsymbol{\Omega}_k$ will be proportional to the values of the hidden units $\mathbf{h}_k$ that they multiply. Recall that we already computed these during the forward pass.

Problem 7.7

7.2.2 Backpropagation algorithm summary

We'll now briefly summarize the final backpropagation algorithm. Consider a deep neural network $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that takes input $\mathbf{x}_i$, has K hidden layers with ReLU activations, and individual loss term $l_i = \mathrm{l}[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], \mathbf{y}_i]$. The goal of backpropagation is to compute the derivatives $\partial l_i/\partial \boldsymbol{\beta}_k$ and $\partial l_i/\partial \boldsymbol{\Omega}_k$ with respect to the weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$.

Forward pass: We compute and store the following quantities:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] & k \in \{1, 2, \ldots K\} \\
\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k. & k \in \{1, 2, \ldots K\}
\end{aligned} \tag{7.12}$$

Backward pass: We start with the derivative $\partial l_i/\partial \mathbf{f}_K$ of the loss function l_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned}
\frac{\partial l_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial l_i}{\partial \mathbf{f}_k} & k \in \{K, K-1, \ldots 1\} \\
\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial l_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T & k \in \{K, K-1, \ldots 1\} \\
\frac{\partial l_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial l_i}{\partial \mathbf{f}_k} \right), & k \in \{K, K-1, \ldots 1\}
\end{aligned} \tag{7.13}$$

where $\odot$ denotes pointwise multiplication and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned}
\frac{\partial l_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial l_i}{\partial \mathbf{f}_0} \\
\frac{\partial l_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial l_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T.
\end{aligned} \tag{7.14}$$

Problems 7.8-7.9

We calculate these derivatives for every training example in the batch and sum them together to retrieve the gradient for the stochastic gradient descent update.

7.2.3 Algorithmic differentiation

Although it's important to understand the backpropagation algorithm, it's unlikely that you will need to code it in practice. Modern deep learning frameworks such as PyTorch and Tensorflow calculate the derivatives automatically given the model specification. This is known as *algorithmic differentiation*.

Each functional component (linear transform, ReLU activation, loss function) in the framework knows how to compute its own derivative. For example, the PyTorch ReLU function $\mathbf{z}_{out} = \textbf{relu}[\mathbf{z}_{in}]$ knows how to compute the derivative of its output $\mathbf{z}_{out}$ with respect to its input $\mathbf{z}_{in}$. Similarly, a linear function $\mathbf{z}_{out} = \boldsymbol{\beta}+\boldsymbol{\Omega}\mathbf{z}_{in}$ knows both how to compute the derivatives of the output $\mathbf{z}_{out}$ with respect to the input $\mathbf{z}_{in}$ and with respect to the parameters $\boldsymbol{\beta}$ and $\boldsymbol{\Omega}$. The algorithmic differentiation framework also knows the sequence of operations in the network and thus has all the information required to perform the forward and backward passes.

These frameworks exploit the massive parallelism of modern graphic processor units (GPUs). Computations such as matrix multiplication (which features in both the forward and backward pass) are naturally amenable to parallelization. Moreover, it's possible to perform the forward and backward passes for all of the examples in the batch in parallel, if the model and the intermediate results in the forward pass do not exceed the available memory.

Problem 7.10

Since the training algorithm now processes the entire batch in parallel, the input becomes a multi-dimensional *tensor*. In this context, a tensor can be considered the generalization of a matrix to arbitrary dimensions. Hence, a vector is a 1D tensor, a matrix is a 2D tensor, and a 3D tensor is a three-dimensional grid of numbers. Until now, the training data have been one-dimensional, and so the input for backpropagation would be a 2D tensor where the first dimension is the data index and the second is the batch index. In subsequent chapters, we will encounter more complex structured input data. For example, in models where the input is an RGB image, the original data examples are three-dimensional (height $\times$ width $\times$ channel). Here, the input to the learning framework would be a 4D tensor, where the last dimension indexes the batch element.

7.2.4 Extension to arbitrary computational graphs

In the previous section, we described backpropagation in a deep neural network; this model is a sequence of computational steps in which we calculate the intermediate quantities $\mathbf{f}_0, \mathbf{h}_1, \mathbf{g}_1, \mathbf{h}_2 \ldots \mathbf{f}_k$ in turn. However, models need not be restricted to sequential computation. Later in this book, we will meet models with branching structures. For example, we might take the values in a hidden layer and process them through two different sub-networks before recombining.

Problems 7.11-7.12

Fortunately, the ideas of backpropagation still hold if the computational graph is acyclic. Modern algorithmic differentiation frameworks such as PyTorch and Tensorflow can handle arbitrary acyclic computational graphs.

7.3 Parameter initialization

We have discussed both stochastic gradient descent, and how to compute the derivatives that it requires. We now address how to initialize the parameters before we start training. To see why this is important, consider that during the forward pass, each hidden layer $\mathbf{h}_k$ is computed as:

$$\mathbf{h}_{k+1} = \mathbf{a}[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k],$$

where $\mathbf{a}[\bullet]$ applies the ReLU functions and $\boldsymbol{\Omega}_k$ and $\boldsymbol{\beta}_k$ are the weights and biases respectively. Imagine that we initialize all the biases to zero, and the elements of $\boldsymbol{\Omega}_k$ according to a normal distribution with mean zero and variance σ^2. Let's now consider two scenarios:

- If the variance σ^2 is very small (*e.g.*, 10^{-5}), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very small; the result is likely to have a smaller magnitude than the input. After passing through the ReLU function, values less than zero will be clipped and the range of outputs will halve. Consequently, the values at the hidden layers will tend to get smaller and smaller as we progress through the network.

- If the variance σ^2 is very large (*e.g.*, 10^5), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very large; the result is likely to have a much larger magnitude than the input. After passing through the ReLU function, any values less than zero will be clipped and so the range of outputs will be halved; however, even after this, their magnitude might still be larger on average. Consequently, the values at the hidden layers will tend to get larger and larger as we progress through the network.

In these two situations, the values at the hidden layers may become so small or so large that they cannot be represented with finite precision floating point math.

Even if the forward pass is tractable, the same logic applies to the backward pass. Each gradient update (equation 7.13) consists of multiplying by $\boldsymbol{\Omega}^T$, and if the values of $\boldsymbol{\Omega}$ are not initialized sensibly, then the gradient magnitudes may decrease or increase uncontrollably as we perform the backward pass. These cases are respectively known as the *vanishing gradient problem* and the *exploding gradient problem*. In the former case, updates to the model become vanishingly small. In the latter case, the updates become unstable.

Problem 7.13

7.3.1 Initialization for forward pass

We now present a mathematical version of the same argument. Consider the computation between adjacent hidden layers $\mathbf{h}$ and $\mathbf{h}'$ of a neural network, which have dimensions D_h and $D_{h'}$ respectively:

$$\begin{aligned} \mathbf{f} &= \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h} \\ \mathbf{h}' &= \mathbf{a}[\mathbf{f}], \end{aligned}$$

where $\mathbf{f}$ represents the pre-activations, $\boldsymbol{\Omega}$, and $\boldsymbol{\beta}$ represent the weights and biases, and $\mathbf{a}[\bullet]$ is the activation function.

Assume the hidden units h_j in the input layer $\mathbf{h}$ have variance σ^2_h. Consider initializing the biases β_i as zero, and the weights Ω_{ij} as normally distributed with mean zero and variance σ^2_Ω. Now we'll derive expressions for the mean and variance of the intermediate values $\mathbf{f}$ and the subsequent hidden layer $\mathbf{h}$.

The mean $\mathbb{E}[f_i]$ of the intermediate values f_i are:

Expectation operator Appendix B.2

$$\begin{aligned}\mathbb{E}[f_i] &= \mathbb{E}\left[\beta_i+\sum_{j=1}^{D_h}\Omega_{ij}h_j\right]\\ &= \mathbb{E}\left[\beta_i\right]+\sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}h_j\right]\\ &= \mathbb{E}\left[\beta_i\right]+\sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}\right]\mathbb{E}\left[h_j\right]\\ &= 0+\sum_{j=1}^{D_h}0\cdot\mathbb{E}\left[h_j\right]=0,\end{aligned} \tag{7.15}$$

where D_h is the dimensionality of the input layer $\mathbf{h}$ and we have assumed that the distributions over the hidden units h_j and the network weights Ω_{ij} are independent between the second and third lines.

Using this result, we see that the variance σ^2_f of the intermediate values f_i is:

$$\begin{aligned}\sigma^2_f &= \mathbb{E}[f_i^2]-\mathbb{E}[f_i]^2\\ &= \mathbb{E}\left[\left(\sum_{j=1}^{D_h}\Omega_{ij}h_j\right)^2\right]-0\\ &= \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}^2\right]\mathbb{E}\left[h_j^2\right]\\ &= \sum_{j=1}^{D_h}(\sigma^2_\Omega\sigma^2_h)=D_h\sigma^2_\Omega\sigma^2_h,\end{aligned} \tag{7.16}$$

where we have used the standard identity $\sigma^2=\mathbb{E}[(z-\mathbb{E}[z])^2]=\mathbb{E}[z^2]-\mathbb{E}[z]^2$. We have assumed once more that the distributions of the weights Ω_{ij} and the hidden units h_j are independent between lines two and three.

Variance identity Appendix B.2.2

The initial distribution of the weights Ω_{ij} was symmetric about zero, and so the distribution of f_j will also be symmetric about zero. It follows that half of the weights will be clipped by the ReLU function and so the variance of $\mathbf{h}'$ will be half the variance of $\mathbf{f}$:

Problem 7.14

$$\sigma^2_{h'}=\frac{1}{2}\sigma^2_f=\frac{1}{2}D_h\sigma^2_\Omega\sigma^2_h. \tag{7.17}$$

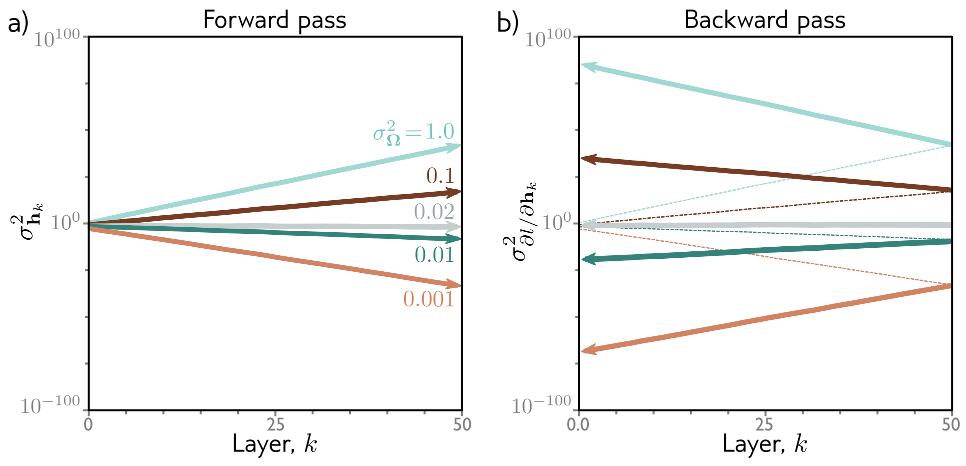


Figure 7.4 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100 dimensional input $\mathbf{x}$ initialized with values from a standard normal distribution, a single output fixed at $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero and the weight matrices $\mathbf{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_{\mathbf{\Omega}} \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a) Variance of hidden unit activations computed in forward pass as a function of the network layer. For He initialization ($\sigma^2_{\mathbf{\Omega}} = 2/D_h = 0.02$), the variance is stable. However, for larger values it increases rapidly, and for smaller values, it decreases rapidly. b) The variance of the gradients in the backward pass (solid lines) continues this trend; if we initialize with a value larger than 0.02, the magnitude of the gradients increases rapidly as we pass back through the network. If we initialize with a value smaller, then the magnitude decreases. These are respectively known as the *exploding gradient* and *vanishing gradient* problems.

This in turn implies that if we want the variance $\sigma^2_{h'}$ of the subsequent layer $\mathbf{h}'$ to be the same as the variance of the original layer $\mathbf{h}$ during the forward pass, we should set:

$$\sigma^2_{\Omega} = \frac{2}{D_h}, \tag{7.18}$$

where D_h is the dimension of the original layer that the weights were applied to. This is known as *He initialization.*

7.3.2 Initialization for backward pass

A similar argument establishes how the variance of the gradients $\partial l/\partial f_k$ changes during the backward pass. During the backward pass, we multiply by the transpose $\mathbf{\Omega}^T$ of the weight matrix (equation 7.13) and so the equivalent expression becomes:

$$\sigma_\Omega^2 = \frac{2}{D_{h'}}, \tag{7.19}$$

where $D_{h'}$ is the dimension of the layer that the weights feed into.

7.3.3 Initialization for both forward and backward pass

If the weight matrix $\mathbf{\Omega}$ is not square (*i.e.*, there are different numbers of hidden units in the two adjacent layers and so D_h and $D_{h'}$ differ), then it is not possible to choose the variance to satisfy both equations 7.18 and 7.19 simultaneously. One possible compromise is to use the mean $(D_h + D_{h'})/2$ as a proxy for the number of terms which gives:

$$\sigma_\Omega^2 = \frac{1}{D_h + D_{h'}}. \tag{7.20}$$

Figure 7.4 shows empirically that both the variance of the hidden units in the forward pass and the variance of the gradients in the backward pass remain stable when the parameters are initialized appropriately.

7.4 Example training code

The primary focus of this book is scientific; this is not a guide for how to implement deep learning models. Nonetheless, in figure 7.5 we present working PyTorch code that implements the ideas explored in this book so far. The code defines a neural network and initializes the weights. It creates a random set of input and output data and defines a least squares loss function. The model is trained from the data using stochastic gradient descent with momentum in batches of size 10 over 100 epochs. The learning rate starts at 0.01 and halves every 10 epochs.

Problems 7.15-7.16

The takeaway is that although the underlying ideas in deep learning are quite complex, implementation is relatively simple. For example, all of the details of the backpropagation are hidden in the single line of code: `loss.backward()`.

7.5 Summary

In the previous chapter, we introduced stochastic gradient descent, which is an iterative optimization algorithm that aims to find the minimum of a function. In the context of neural networks, we apply this algorithm to find the parameters that minimize the loss function.

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
   nn.Linear(D_i, D_k),
   nn.ReLU(),
   nn.Linear(D_k, D_k),
   nn.ReLU(),
   nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
   if isinstance(layer_in, nn.Linear):
   nn.init.kaiming_uniform(layer_in.weight)
   layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 dummy data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

 # loop over the dataset 10 times
for epoch in range(100):
   epoch_loss = 0.0
   # loop over batches
   for i, data in enumerate(data_loader):
   # retrieve inputs and labels for this batch
   x_batch, y_batch = data
   # zero the parameter gradients
   optimizer.zero_grad()
   # forward pass
   pred = model(x_batch)
   loss = criterion(pred, y_batch)
   # backward pass
   loss.backward()
   # SGD update
   optimizer.step()
   # update statistics
   epoch_loss += loss.item()
   # print error
   print('Epoch %5d, loss: %.3f' %(epoch, epoch_loss))
   # tell scheduler to consider updating learning rate
   scheduler.step()
```

Figure 7.5 Sample code for training two-layer network on random data.

Stochastic gradient descent requires that we compute the gradient of the loss function with respect to the parameters and that we initialize these parameters before optimization. In this chapter, we have addressed these two problems for deep neural networks. The gradients must be evaluated for a very large number of parameters, for each member of the batch, and at each SGD iteration. It is hence imperative that the gradient computation is efficient and to this end, we introduced the backpropagation algorithm.

We also saw that parameter initialization is critical. The magnitudes of the hidden unit activations can either decrease or increase exponentially in the forward pass. The same is true of the gradient magnitudes in the backward pass where these behaviors are known as the disappearing gradient and exploding gradient problems. Both impede training but can be avoided with appropriate initialization.

We've now reached a point where we have defined the model and the loss function and can train a model for a given task. The following chapter discusses how to measure the performance of that model.

Notes

Backpropagation: Efficient re-use of partial computations while calculating gradients in computational graphs has been repeatedly discovered, including by Werbos (1974), Bryson *et al.* (1979), Lecun (1985), and Parker (1985). However, the most celebrated description of this idea was by Rumelhart *et al.* (1985) and Rumelhart *et al.* (1986) who also coined the term 'backpropagation'. This latter work kick-started a new phase of neural network research in the eighties and nineties; for the first time, it was practical to train networks with hidden layers. However, progress stalled due (in retrospect) to a lack of training data, limited computational power, and the use of sigmoid activations. Applied areas of machine learning such as natural language processing and computer vision still relied on non-neural network methods until the remarkable image classification results of Krizhevsky *et al.* (2012) ushered in the modern era of deep learning.

The implementation of backpropagation in modern deep learning frameworks such as PyTorch and Tensorflow is an example of reverse mode algorithmic differentiation. This is distinguished from forward mode algorithmic differentiation in which the derivatives from the chain rule are accumulated while moving forward through the computational graph (see problem 7.12). Further information about algorithmic differentiation can be found in Griewank & Walther (2008) and Baydin *et al.* (2018).

Initialization: He initialization was first introduced in He *et al.* (2015). It follows closely from *Glorot* or *Xavier* initialization (Glorot & Bengio 2010), which is very similar but does not consider the effect of the ReLU layer and so differs by a factor of two. Essentially the same method was proposed much earlier by LeCun *et al.* (1998b) but with a slightly different motivation; in this case, sigmoidal activation functions were used, which naturally normalize the range of outputs at each layer and hence help prevent an exponential increase in the magnitudes of the hidden units. However, if the inputs are too large, then they fall into the flat regions of the sigmoid function and result in very small gradients. Hence, it is still important to initialize the weights sensibly. Klambauer *et al.* (2017) introduce the scaled exponential unit and show that within a certain range of inputs, this activation function tends to make the activations in network layers automatically converge to mean zero and unit variance.

A completely different approach is to pass data through the network, and then normalize by the empirically observed variance. *Layer-sequential unit variance initialization* is an example of this kind of method, in which the weight matrices as initialized as orthonormal. GradInit (Zhu *et al.* 2021) randomizes the initial weights and temporarily fixes them while it learns non-negative scaling factors for each weight matrix. These factors are selected so that they maximize the decrease in the loss for a fixed learning rate subject to a constraint on the maximum norm of the gradient. Closely related to these methods are schemes such as *BatchNorm* (Ioffe & Szegedy 2015), in which the network normalizes the variance of each batch as part of its processing at every step. BatchNorm and its variants are discussed in chapter 11. Other initialization schemes have been proposed for specific architectures, including the *ConvolutionOrthogonal* initializer (Xiao *et al.* 2018) for convolutional networks, *Fixup* (Zhang *et al.* 2019) for residual networks and *TFixup* (Huang *et al.* 2020), and *DTFixup* (Xu *et al.* 2021) for transformers.

Reducing memory requirements: Training neural networks is demanding in terms of memory. Not only must we store the model parameters, but we must also store the pre-activations at the hidden units for every member of the batch during the forward pass. Two methods that decrease memory requirements are *gradient checkpointing* (Chen *et al.* 2016) and *micro-batching* (Huang *et al.* 2019). In gradient checkpointing, the activations are only stored every N layers during the forward pass. During the backward pass, the intermediate missing activations are recalculated from the nearest checkpoint. In this manner, we can drastically reduce the memory requirements at the computational cost of performing the forward pass twice (problem 7.10). In micro-batching, the batch is sub-divided into smaller parts and the gradient updates are aggregated from each sub-batch before being applied to the network. A completely different approach is to build a reversible network (*e.g.*, (Gomez *et al.* 2017)), in which the previous layer can be computed from the current one and so there is no need to cache anything during the forward pass (see chapter 15). These methods and other approaches to reducing memory requirements are reviewed in Sohoni *et al.* (2019).

Distributed training: For sufficiently large models, the memory requirements or total required time may be too much for a single processor. In this case, we must use *distributed training*, in which training takes place in parallel across multiple processors. There are several approaches to parallelism. In *data parallelism*, each processor or *node* contains a full copy of the model but runs a subset of the batch (see Xing *et al.* 2015, Li *et al.* 2020). The gradients from each node are aggregated centrally and then redistributed back to each node to ensure that the models remain consistent. This is known as *synchronous training.* The synchronization required to aggregate and redistribute the gradients can be a performance bottleneck and this leads to the idea of asynchronous training. For example, in the Hogwild! algorithm (Niu *et al.* 2011) the gradient from a node is used to update a central model whenever it is ready. The updated model is then redistributed to the node. This means that each node may have a slightly different version of the model at any given time and so the gradient updates may be stale; however, it works well in practice. Other, decentralized schemes have also been developed. For example, in Zhang *et al.* (2016b), the individual nodes update one another in a ring structure.

Data parallelism methods still assume that the entire model can be held in the memory of a single node. *Pipeline model parallelism* stores different layers of the network on different nodes and so does not have this requirement. In a naïve implementation, the first node runs the forward pass for the batch on the first few layers and passes the result to the next node, which runs the forward pass on the next few layers and so on. In the backward pass, the gradients are updated in the opposite order. The obvious disadvantage of this approach is that each machine lies idle for most of the cycle. Various schemes revolving around each node processing micro-batches sequentially have been proposed to reduce this inefficiency (*e.g.*, Huang *et al.* 2019, Narayanan *et al.* 2021a). Finally, in *tensor model* *parallelism*, the computation at a single network layer is distributed across nodes (*e.g.*, Shoeybi *et al.* 2020). A good overview of distributed training methods can be found in Narayanan *et al.* (2021b) who combine tensor, pipeline, and data parallelism to train a language model with one trillion parameters on 3072 GPUs.

Problems

Problem 7.1 A two-layer network with two hidden units in each layer can be defined as:

$$\begin{aligned} y \quad = \quad & \phi_0 + \phi_1 \mathrm{a}\left[\psi_{01} + \psi_{11}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{21}\mathrm{a}[\theta_{02} + \theta_{12}x]\right] \\ & \qquad + \phi_2 \mathrm{a}[\psi_{02} + \psi_{12}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{02} + \theta_{12}x]], \end{aligned} \tag{7.21}$$

where the functions $a[\bullet]$ are ReLU functions Compute the derivatives of the output y with respect to each of the 15 parameters $\boldsymbol{\phi}_\bullet, \theta_{\bullet\bullet}$, and $\psi_{\bullet\bullet}$ directly (*i.e.*, not using the backpropagation algorithm). The derivative of the ReLU function with respect to its input $\partial a[z]/\partial z$ is the indicator function $\mathbb{I}[z > 0]$, which returns one if the argument is greater than zero and zero otherwise (figure 7.3).

Problem 7.2 Calculate the derivative $\partial l_i / \partial f[\mathbf{x}_i, \boldsymbol{\phi}]$ for the least squares loss function:

$$l_i = (y - \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2. \tag{7.22}$$

Problem 7.3 Calculate the derivative $\partial l_i / \partial f[\mathbf{x}_i, \boldsymbol{\phi}]$ for the binary classification loss function:

$$l_i = -\sum_{i=1}^{I} (1 - y_i) \log\left[1 - \mathrm{sig}[\mathrm{f}[\mathbf{x}_i; \boldsymbol{\phi}]]\right] + y_i \log\left[\mathrm{sig}[\mathrm{f}[\mathbf{x}_i; \boldsymbol{\phi}]]\right], \tag{7.23}$$

where the function $\mathrm{sig}[\bullet]$ is the logistic sigmoid and is defined as:

$$\mathrm{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{7.24}$$

Problem 7.4 Show that:

$$\frac{\partial}{\partial \mathbf{h}} (\boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}) = \boldsymbol{\Omega}^T, \tag{7.25}$$

where $\mathbf{h}$ and $\boldsymbol{\beta}$ are $D \times 1$ vectors and $\boldsymbol{\Omega}$ is a $D \times D$ matrix, and $\partial/\partial\mathbf{h}$ is the vector $[\partial/\partial h_1, \partial/\partial h_1, \ldots, \partial/\partial h_D]^T$. To accomplish this, re-write the expression $\boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$ in terms of sums of the constituent elements (*e.g.*, so $\mathbf{Ab} = \sum_j a_{ij} b_j$), take the derivatives as normal, and then rewrite in matrix form.

Problem 7.5 Consider the case where we use the logistic sigmoid (see equation 7.24) as an activation function so $h = \mathrm{sig}[f]$. Compute the derivative $\partial h / \partial f$ for this activation function. What happens to the derivative when the input takes (i) a large positive value and (ii) a large negative value?

Problem 7.6 Consider using (i) the Heaviside function and (ii) the rectangular function as activation functions:

$$\text{Heaviside}[z] = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}, \tag{7.26}$$

and

$$\text{rect}[z] = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}. \tag{7.27}$$

Discuss why these functions are problematic for neural network training with gradient-based optimization methods.

Problem 7.7 Show that:

$$\frac{\partial}{\partial \boldsymbol{\Omega}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k\right) \frac{\partial l_i}{\partial \mathbf{f}_k} = \frac{\partial l_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T, \tag{7.28}$$

where $\partial/\partial\boldsymbol{\Omega}$ is a matrix where element (i,j) is given by $\partial/\partial\Omega_{ij}$. The terms $\boldsymbol{\beta}_k$, $\boldsymbol{\Omega}_k$, and $\mathbf{h}_k$ have sizes $D_{k+1} \times 1$, $D_{k+1} \times D_k$, and $D_k \times 1$ respectively. The term $\partial l_i/\partial \mathbf{f}_k$ is also of size $D_k \times 1$. As in problem 7.4, the easiest way to show this is to compute each term $\partial/\partial\Omega_{ij}$ separately, and then re-write in matrix form.

Problem 7.8 Write native Python code (*i.e.*, *not* using PyTorch or Tensorflow) that implements the forward and backward passes of the backpropagation algorithm for a neural network with one input, one output, K hidden layers with D neurons each, and a least squares loss function. Assume that there are 100 input examples drawn from a standard normal distribution, each of which maps to an output that is also drawn from a standard normal distribution, and that we are calculating the derivatives for the entire dataset (rather than a batch). Initialize the biases to all zeros and the weights from a standard normal distribution. How could you test whether the derivatives that you calculate are correct for a known input/output pair $\{x, y\}$?

Problem 7.9 Derive the equations for the backward pass of the backpropagation algorithm for a network that uses leaky ReLU activations, which are defined as:

$$\text{a}[z] = \text{ReLU}[z] = \begin{cases} \alpha z & z < 0 \\ z & z \geq 0 \end{cases}, \tag{7.29}$$

where α is a small positive constant (typically 0.1).

Problem 7.10 Consider the situation where we only have enough memory to store the values at every tenth hidden layer during the forward pass. Adapt the backpropagation algorithm to compute the derivatives in this situation using gradient checkpointing.

Problem 7.11 This problem explores computing derivatives on general acyclic computational graphs. Consider the function:

$$y = \exp\left[\exp[x] + \exp[x]^2\right] + \sin[\exp[x] + \exp[x]^2]. \tag{7.30}$$

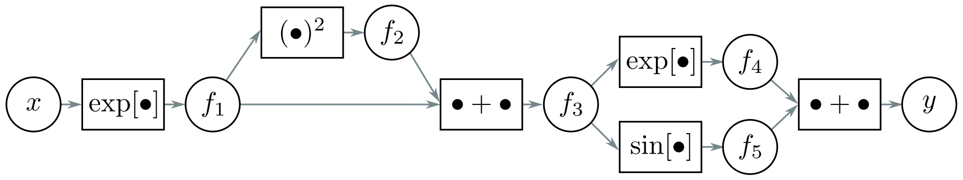


Figure 7.6 Computational graph for problems 7.11 and 7.12. Adapted from Domke (2010).

We can break this down into a series of intermediate computations so that:

$$\begin{aligned} f_1 &= \exp[x] \\ f_2 &= f_1^2 \\ f_3 &= f_1 + f_2 \\ f_4 &= \exp[f_3] \\ f_5 &= \sin[f_3] \\ y &= f_4 + f_5. \end{aligned} \tag{7.31}$$

The associated computational graph is depicted in figure 7.6. Compute the derivative $\partial y/\partial x$, by *reverse mode differentiation*. In other words, compute in order:

$$\frac{\partial y}{\partial f_5}, \frac{\partial y}{\partial f_4}, \frac{\partial y}{\partial f_3}, \frac{\partial y}{\partial f_2}, \frac{\partial y}{\partial f_1} \text{ and } \frac{\partial y}{\partial x}, \tag{7.32}$$

using the chain rule in each case to make use of the derivatives already computed.

Problem 7.12 For the same function in problem 7.30, compute the derivative $\partial y/\partial x$, by *forward mode differentiation*. In other words, compute in order:

$$\frac{\partial f_1}{\partial x}, \frac{\partial f_2}{\partial x}, \frac{\partial f_3}{\partial x}, \frac{\partial f_4}{\partial x}, \frac{\partial f_5}{\partial x}, \text{ and } \frac{\partial y}{\partial x}, \tag{7.33}$$

using the chain rule in each case to make use of the derivatives already computed. Why do we not use forward mode differentiation when we compute the parameter gradients for deep networks?

Problem 7.13 Use your code from problem 7.8 to recreate figure 7.4 showing the phenomena of vanishing and exploding gradients.

Problem 7.14 Consider a set of variables z_d with mean $\mathbb{E}[z_d] = 0$ and variance $\text{Var}[z_d] = \sigma^2$. Show that if we pass these variables through the ReLU function:

$$z'_d = \text{ReLU}[z_d] = \begin{cases} 0 & z_d < 0 \\ z_d & z_d \geq 0 \end{cases}. \tag{7.34}$$

then the variance of the transformed variables $\text{Var}[z'_d] = \sigma^2/2$.

Problem 7.15 Implement the code in figure 7.5 in PyTorch and plot the batch loss as a function of the number of iterations.

Problem 7.16 Change the code in figure 7.5 to tackle a binary classification problem. You will need to (i) change the targets y so that they are binary and (ii) change the loss function appropriately.

Chapter 8

Measuring performance

Previous chapters described neural network models, loss functions, and training algorithms. This chapter considers how to measure the performance of the trained models. With sufficient capacity (*i.e.*, number of hidden units), a neural network model will often perform perfectly on the training data. However, this does not necessarily mean that it will generalize well to new test data.

We will see that the test errors have three distinct causes and that their relative contributions depend on (i) the inherent uncertainty in the task, (ii) the amount of training data, and (iii) the model capacity. The latter dependency raises the issue of hyperparameter search. We discuss how to select both the model hyperparameters (*e.g.*, the number of hidden layers and the number of neurons in each) and the learning algorithm hyperparameters (*e.g.*, the learning rate and batch size).

8.1 Training a simple model

We'll explore model performance using the MNIST-1D dataset (figure 8.1). This consists of ten classes $y \in \{0, 1, \ldots 9\}$, representing the digits 0-9. The data are derived from 1D templates for each of the digits. Each data example $\mathbf{x}$ is created by choosing a template, randomly transforming this template, and finally adding noise. The full training dataset $\{\mathbf{x}_i, y_i\}$ consists of $I = 4000$ training examples, each of which consists of $D_i = 40$ dimensions representing the 40 sampled horizontal positions. The ten classes are drawn uniformly during data generation and so there are ~ 400 examples of each class.

We use a neural network with $D_i = 40$ inputs and $D_o = 10$ outputs which are passed through the softmax function to produce class probabilities (see section 5.6). The network has two hidden layers with $D = 100$ hidden units each. It is trained with a multi-class cross-entropy loss using SGD with batch size 100 and learning rate 0.1 for 6000 steps (150 epochs). Figure 8.2 shows that the training error decreases as training proceeds. The training data are classified perfectly after about 4000 steps. The training loss also decreases, eventually approaching zero. Problem 8.1

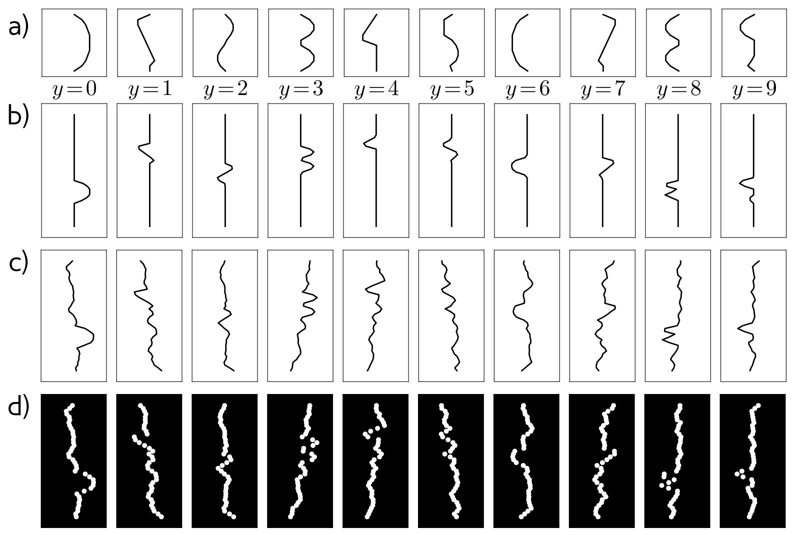


Figure 8.1 MNIST-1D (Greydanus 2020). a) Templates for 10 classes $y \in \{0, \ldots 9\}$, based on digits 0-9. b) Training examples $\mathbf{x}$ are created by randomly transforming a template and c) adding noise. d) The horizontal position of the transformed template is then sampled at 40 positions.

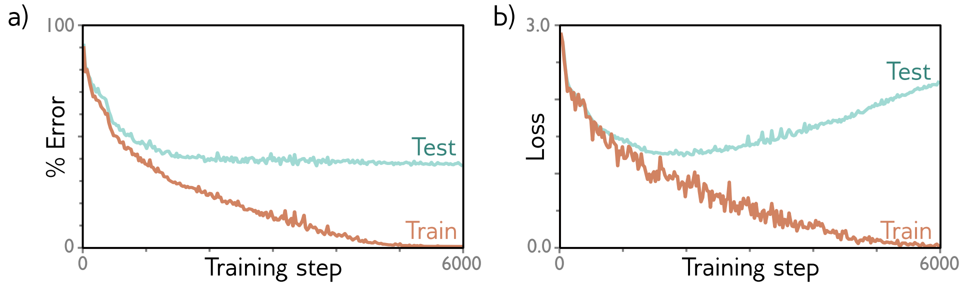


Figure 8.2 MNIST1D results. a) Percent classification error as a function of the training step. The training set errors decrease to zero, but the test errors do not decrease below $\sim 40\%$. This model doesn't generalize well to new test data. b) Loss as a function of the training step. The training loss decreases steadily towards zero. The test loss decreases at first but then increases as the model becomes increasingly confident about its (wrong) predictions.

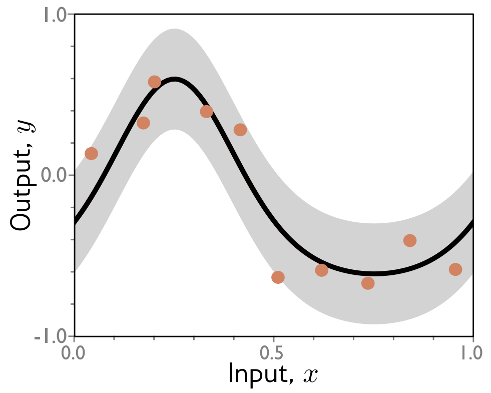


Figure 8.3 Regression function. Solid black line shows the ground truth function. To generate I training examples $\{x_i, y_i\}$ we divide the input space $x \in [0,1]$ into I equal segments and sample one x_i from a uniform distribution within each segment. The corresponding value y_i is created by evaluating the function at x_i and adding Gaussian noise (gray region shows ± 2 standard deviations). The test data are generated in the same way.

However, this doesn't imply that the classifier is perfect; the model might just have memorized the training set but have no idea how to predict new examples. To estimate the true performance, we need a separate *test set* of input/output pairs $\{\mathbf{x}_i, y_i\}$. To this end, we generate 1000 more examples using the same process. Figure 8.2a also shows the errors for this test data as a function of the training step. These decrease as training proceeds, but only to around 40%. This is better than the 90% error rate we would expect by chance but far worse than for the training set; the model has not *generalized* well to the test data.

The test loss (figure 8.2b) decreases for the first 1500 training steps but then increases again. At this point, the test error rate is fairly constant; the model makes the same mistakes, but with increasing confidence. This decreases the probability of the correct answers, and so increases the negative log-likelihood. The increasing confidence is essentially a side-effect of the softmax function; the pre-softmax activations are driven to increasingly extreme values to move the probability of the training examples towards one (see figure 5.9).

8.2 Sources of error

We now consider the sources of the errors that occur when a model fails to generalize. To make this easier to visualize, we'll revert to a 1D linear least squares regression problem where we know exactly how the ground truth data were generated. Figure 8.3 shows a quasi-sinusoidal function; both training and test data are generated by sampling input values in the range $[0, 1]$, passing them through this function and adding Gaussian noise with a fixed standard deviation.

We'll fit a simplified shallow neural net to this data (figure 8.4). The weights and biases that connect the input layer to the hidden layer are chosen so that the 'joints' of the function are evenly spaced across the interval. If there are D hidden units, then these joints will be at $0, 1/D, 2/D, \ldots (D-1)/D$. This model can represent any piecewise linear function with D equally-sized regions in the range $[0, 1]$. As well as being easy to understand, this model also has the advantage that

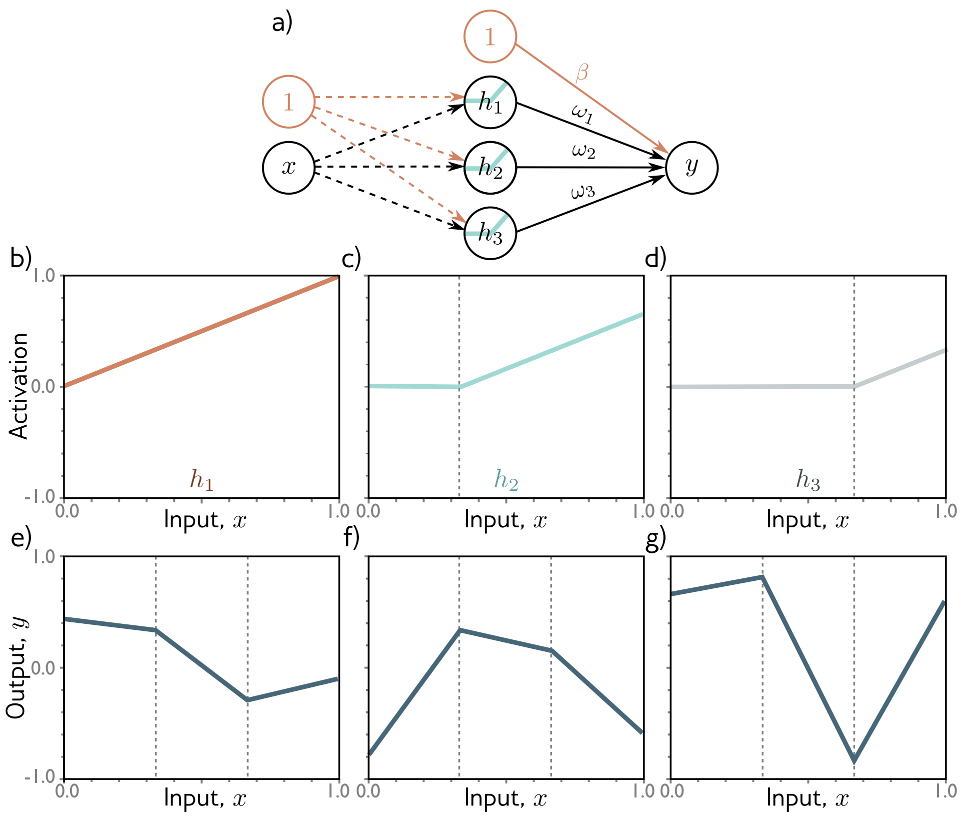


Figure 8.4 Simplified neural network with three hidden units. a) The weights and biases between the input and hidden layer are fixed (dashed arrows). b-d) They are chosen so that the hidden unit activations have slope one and their joints are equally spaced across the interval. With three hidden units the joints are at $x = 0$, $x = 1/3$ and $x = 2/3$ respectively. e-g). This leaves four free parameters $\boldsymbol{\phi} = \{\beta, \omega_1, \omega_2, \omega_3\}$. This model can describe any piecewise linear function over $x \in [0, 1]$ with joints at $1/3$ and $2/3$.

it can be fit in closed form without the need for stochastic optimization algorithms (see problem 8.3). Consequently, we can guarantee to find the global minimum of the loss function during training.

Problems 8.2-8.3

8.2.1 Noise, bias, and variance

There are three possible sources of error, which are known as *noise*, *bias*, and *variance* respectively (figure 8.5):

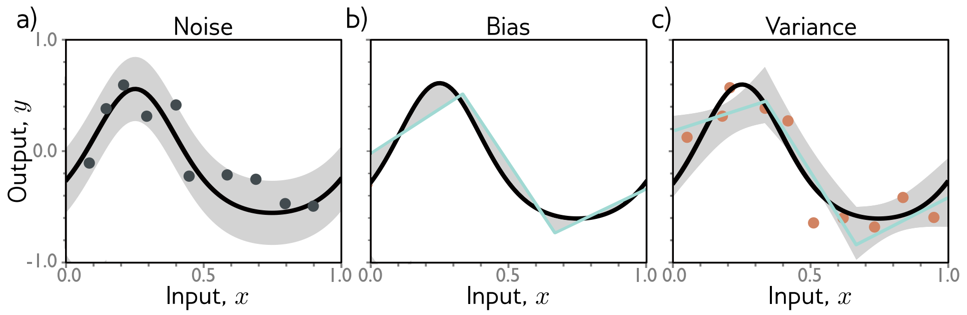


Figure 8.5 Sources of test error. a) Noise. Data generation is noisy, so even if the model exactly replicates the true underlying function (black line), the noise in the test data (gray points) means that some error will remain (gray region represents two standard deviations). b) Bias. Even with the best possible parameters, the three-region model (brown line) cannot fit the true function (black line) exactly. This bias is another source of error (gray regions represent signed error). c) Variance. In practice, we have limited noisy training data (orange points). When we fit the model, we don't recover the best possible function from panel (b) but a slightly different function (cyan line) that reflects idiosyncrasies of the training data. This provides a further source of error (gray region represents two standard deviations). Figure 8.6 shows how this region was calculated.

Noise: The data generation process includes the addition of noise. This means that there are multiple possible valid outputs y for each input x (figure 8.5a). This source of error is insurmountable for the test data. Note that it does not necessarily limit the training performance; during training, it is likely that we never see the same input x twice, so it is still possible fit the training data perfectly.

Noise may arise because there is a genuine stochastic element to the data generation process, because some of the data are mislabeled, or because there are further explanatory variables that were not observed. In rare cases, noise may be absent; for example, a network might be used to approximate a function that is deterministic but requires significant computation to evaluate. However, noise is usually a fundamental limitation on the possible test performance.

Bias: A second potential source of error may occur because the model is not flexible enough to fit the true function perfectly. For example, the three-region neural network model cannot exactly describe the quasi-sinusoidal function, even when the parameters are chosen optimally (figure 8.5b). This is known as *bias*.

Variance: We have limited training examples and there is no way to distinguish systematic changes in the underlying function from noise. When we fit a model, we do not get the closest possible approximation to the true underlying function. Indeed, for different training datasets, the result will be slightly different each time.

This additional source of variability is termed *variance* (figure 8.5c). In practice, there might also be additional variance due to the stochastic learning algorithm.

8.2.2 Mathematical formulation of test error

We'll now make the notions of noise, bias, and variance mathematically precise. Consider a 1D regression problem where the data generation process has additive noise with variance σ^2 (*e.g.*, figure 8.3); we can observe different outputs y for the same input x and so for each x there is a distribution $Pr(y|x)$ with mean $\mu[x]$:

Appendix B.2 Expectation

$$\mu[x] = \mathbb{E}_y[y[x]] = \int y[x]Pr(y|x)dy, \tag{8.1}$$

and fixed noise $\sigma^2 = \mathbb{E}_y\left[(\mu[x]-y[x])^2\right]$. Here we have used the notation $y[x]$ to specify that we are considering the output y at a given input position x.

Now consider a least squares loss between the model prediction $\mathrm{f}[x,\boldsymbol{\phi}]$ at position x and the observed value $y[x]$ at that position:

$$\begin{aligned} L[x] &= (\mathrm{f}[x,\boldsymbol{\phi}]-y[x])^2 \\ &= ((\mathrm{f}[x,\boldsymbol{\phi}])-\mu[x])+(\mu[x]-y[x]))^2 \\ &= (\mathrm{f}[x,\boldsymbol{\phi}])-\mu[x])^2+2(\mathrm{f}[x,\boldsymbol{\phi}])-\mu[x])(\mu[x]-y[x])+(\mu[x]-y[x])^2, \end{aligned} \tag{8.2}$$

where we have both added and subtracted the mean $\mu[x]$ of the underlying function in the second line and have expanded out the squared term in the third line.

Appendix B.2.1 Expectation rules

The underlying function is stochastic, and so this loss depends on the particular $y[x]$ we observe. The expected loss is:

$$\begin{aligned} &\mathbb{E}_y[L[x]] \\ &\quad= \mathbb{E}_y\left[(\mathrm{f}[x,\boldsymbol{\phi}])-\mu[x])^2+2(\mathrm{f}[x,\boldsymbol{\phi}]-\mu[x])(\mu[x]-y[x])+(\mu[x]-y[x])^2\right] \\ &\quad= (\mathrm{f}[x,\boldsymbol{\phi}])-\mu[x])^2+2(\mathrm{f}[x,\boldsymbol{\phi}]-\mu[x])(\mu[x]-\mathbb{E}_y\left[y[x]\right])+\mathbb{E}_y\left[(\mu[x]-y[x])^2\right] \\ &\quad= (\mathrm{f}[x,\boldsymbol{\phi}])-\mu[x])^2+2(\mathrm{f}[x,\boldsymbol{\phi}]-\mu[x])\cdot 0+\mathbb{E}_y\left[(\mu[x]-y[x])^2\right] \\ &\quad= (\mathrm{f}[x,\boldsymbol{\phi}]-\mu[x])^2+\sigma^2, \end{aligned} \tag{8.3}$$

where in the second line, we have distributed the expectation operator and removed it from terms with no dependence on $y[x]$, and in the third line, we note that the second term is zero, since $\mathbb{E}_y[y[x]] = \mu[x]$. Finally, in the fourth line, we have substituted in the definition of the noise σ^2. We can see that the expected loss has been broken down into two terms; the first term is the squared deviation between the model and the true function mean, and the second term is the noise.

The first term can be further partitioned into bias and variance. The parameters $\boldsymbol{\phi}$ of the model $\mathrm{f}[x,\boldsymbol{\phi}]$ depend on the training dataset $\mathcal{D} = \{x_i, y_i\}$ and so really we should write $\mathrm{f}\left[x,\boldsymbol{\phi}[\mathcal{D}]\right]$. The training dataset is a random sample from the data generation process; with a different sample of training data, we would learn different parameters. The expected model output $f_\mu[x]$ is hence:

$$f_\mu[x] = \mathbb{E}_\mathcal{D}\left[\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]]\right]. \tag{8.4}$$

Returning to the first term of equation 8.3, we add and subtract $\mathrm{f}_\mu[x]$ and expand:

$$\begin{aligned}
&(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mu[x])^2 \\
&\quad = \left((\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x]) + (f_\mu[x] - \mu[x])\right)^2 \\
&\quad = (\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x])^2 + 2(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x])(f_\mu[x] - \mu[x]) + (f_\mu[x] - \mu[x])^2.
\end{aligned} \tag{8.5}$$

We then take the expectation with respect to the training data set $\mathcal{D}$:

$$\mathbb{E}_\mathcal{D}\left[(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mu[x])^2\right] = \mathbb{E}_\mathcal{D}\left[(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x])^2\right] + (f_\mu[x] - \mu[x])^2, \tag{8.6}$$

where we have simplified using similar steps as for equation 8.3. Finally, we substitute this result into equation 8.3:

$$\mathbb{E}_\mathcal{D}\left[\mathbb{E}_y[L[x]]\right] = \underbrace{\mathbb{E}_\mathcal{D}\left[(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x])^2\right]}_{\text{variance}} + \underbrace{(f_\mu[x] - \mu[x])^2}_{\text{bias}} + \underbrace{\sigma^2}_{\text{noise}}. \tag{8.7}$$

This equation says that the expected loss after taking into account the uncertainty in the training data $\mathcal{D}$ and the test data y consists of three components that are summed. The variance is uncertainty due to the particular training dataset we sample. The bias is the systematic deviation of the model from the mean of the function that we are modeling. The noise is the inherent uncertainty in the true mapping from input to output. These three sources of error will be present for any task. They combine additively for linear regression with a least squares loss. However, their interaction can be more complex for other types of problem.

8.3 Reducing error

In the previous section, we saw that test error results from three sources: noise, bias, and variance. The noise component is insurmountable; there is nothing we can do to circumvent this, and it represents a fundamental limit on model performance. However, it is possible to reduce the other two terms.

8.3.1 Reducing variance

Recall that the variance results from limited noisy training data. Fitting the model to two different training sets results in slightly different parameters. It follows that we can reduce the variance by increasing the quantity of training data. This averages out the inherent noise and ensures that the input space is well sampled.

Figure 8.6 shows the effect of training with 6, 10, and 100 samples. For each dataset size, we show the best fitting model for three different training datasets. With only six samples, the fitted function is quite different each time; the variance is large. As we increase the number of samples, the fitted model becomes almost the same each time and the variance reduces. In general, adding training data almost always improves test performance.

8.3.2 Reducing bias

The bias term results from the inability of the model to describe the true underlying function. This suggests that we can reduce this error by making the model more flexible. This is usually done by increasing the model *capacity*, which for neural networks means adding more hidden units and/or hidden layers.

In the simplified model, adding capacity corresponds to adding more hidden units in such a way that the interval $[0, 1]$ is divided into more linear regions. Figures 8.7a-c show that (unsurprisingly) this does indeed reduce the bias; as we increase the number of linear regions to ten, the model becomes flexible enough to fit the true function closely.

8.3.3 Bias-variance trade-off

However, figures 8.7d-f show an unexpected side-effect of increasing the model capacity. For a fixed-size training dataset, the variance term increases as the model capacity increases. Consequently, increasing the model capacity does not necessarily reduce the test error. This is known as the *bias-variance trade-off*.

Figure 8.8 explores this phenomenon. In panels a-c) we fit the simplified model with three linear regions to three different datasets of 15 points. Although the datasets differ, the final model is much the same; the noise in the dataset roughly averages out in each linear region. In panels d-f) we fit a model with ten regions to the same three datasets. This model has more flexibility, but this is disadvantageous; the model certainly fits the data better and the training error will be lower, but much of the extra descriptive power is devoted to modeling the noise. This phenomenon is known as *overfitting*.

We've seen that for a fixed-size training dataset, the bias will decrease, but the variance will increase as we add capacity to the model. This suggests that there is an optimal capacity where the bias is not too large, and the variance is still relatively small. Figure 8.9 shows how these terms vary numerically for the toy model as we increase the capacity, using the data from figure 8.8. For regression models, the total expected error is the sum of the bias and the variance, and this sum is minimized when the model capacity is four (*i.e.*, it has four hidden units and four linear regions).

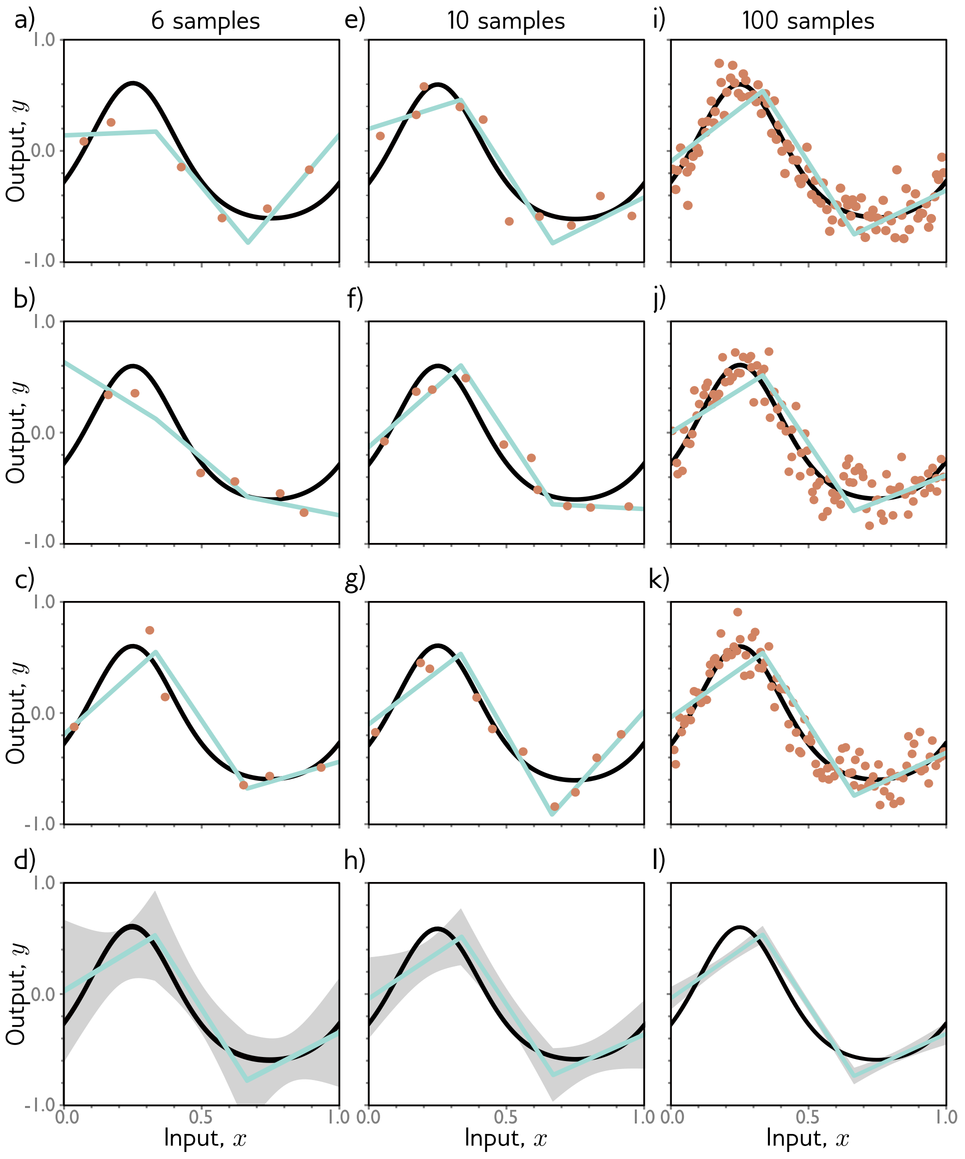


Figure 8.6 Reducing variance by increasing training data. a-c) The three-region model fitted to three different randomly sampled datasets of six points. The fitted model is quite different each time. d) We repeat this experiment many times and plot the mean model predictions (cyan line) and the variance of the model predictions (gray area shows two standard deviations). e-h) We do the same experiment, but this time with a dataset of size ten. The variance of the predictions is reduced. i-l) We repeat this experiment with datasets of size 100. Now the fitted model is always similar, and the variance is small.

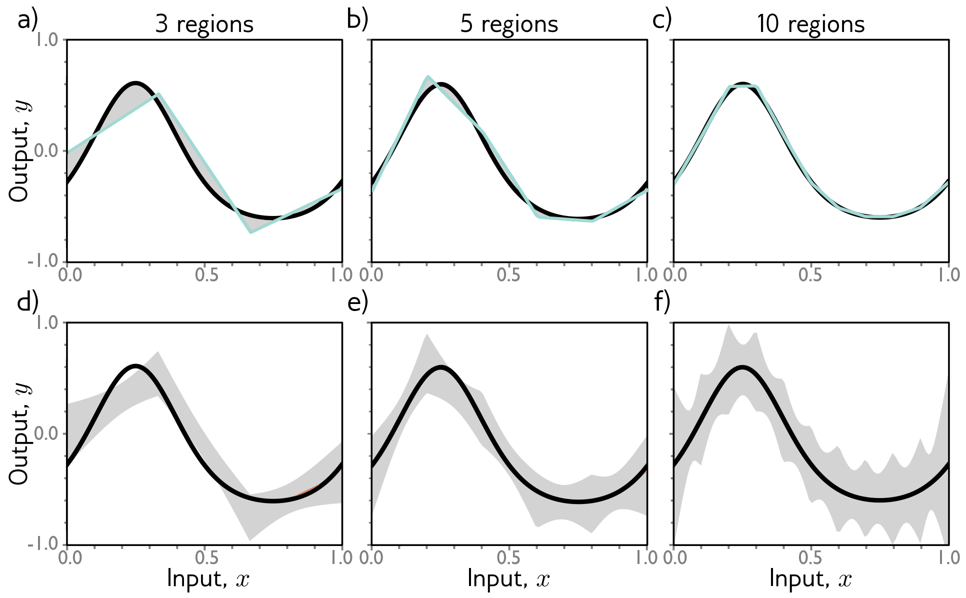


Figure 8.7 Bias and variance as a function of model capacity. a-c) As we increase the number of hidden units of the toy model, the number of linear regions increases, and the model becomes able to fit the true function closely; the bias (gray region) bias decreases. d-f) Unfortunately, increasing the capacity of the model has the side-effect of increasing the variance term (gray region). This is known as the bias-variance trade-off.

8.4 Double descent

In the previous section we examined the bias-variance trade-off as we increase the capacity of a model. Let's now return to the MNIST1D dataset and see whether this happens in practice. We'll train models with 10,000 training examples and test with another 5,000 examples and examine the train and test performance as we increase the capacity (number of parameters) in the model. We trained this model with ADAM and a step size of 0.005 using the full batch of 10,000 examples for 4000 steps.

Figure 8.10a shows the training and test error for a neural network with two hidden layers as the number of hidden units is increased. The training error decreases as the capacity grows and becomes close to zero very quickly. The vertical dashed line represents the capacity where the model has the same number of parameters as there are training examples, but the model memorizes the dataset before this point. The test error decreases as we add model capacity but does not increase as predicted by the bias-variance trade-off curve; it just keeps decreasing.

In figure 8.10b we repeat this experiment, but this time, we randomize 15% of

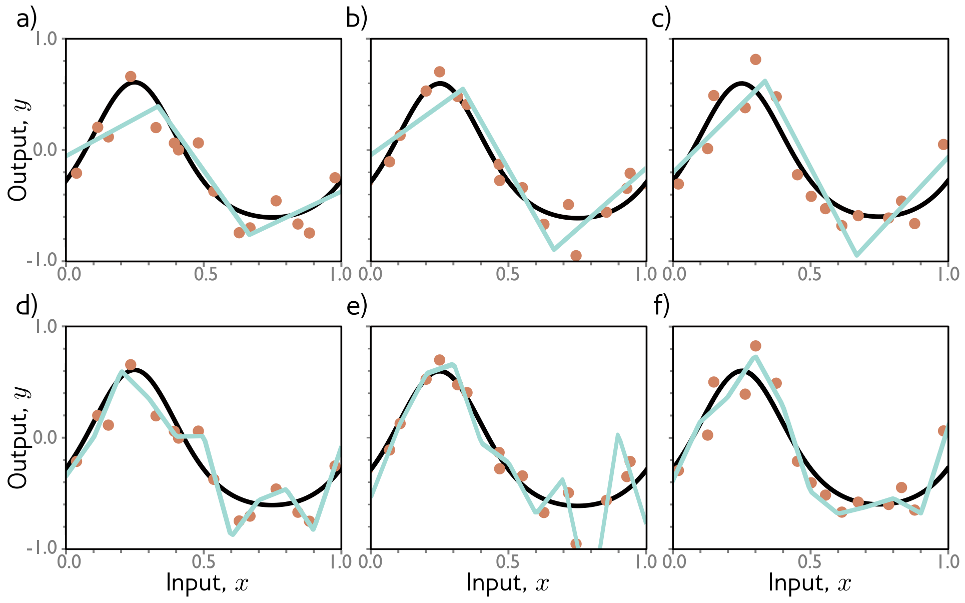


Figure 8.8 Overfitting. a-c) A model with three regions is fit to three different datasets of fifteen points each. The result is very similar in all three cases (*i.e.*, the variance is low). d-f) A model with ten regions is fit to the same datasets. The additional flexibility does not necessarily produce better predictions. While these three models each describe the training data better, they are not necessarily closer to the true underlying function (black curve). Instead, they overfit the data and describe the noise, and the variance (difference between fitted curves) is larger.

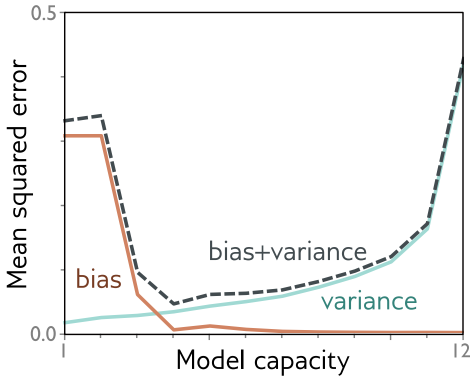


Figure 8.9 Bias-variance trade-off. The bias and variance terms from equation 8.7 are plotted as a function of the model capacity (number of hidden units / linear regions) in the simplified model using training data from figure 8.8. As the capacity increases, the bias (solid orange line) decreases but the variance (solid cyan line) increases. The sum of these two terms (dashed gray line) has a pronounced minimum when the capacity is four.

the training labels. Once more, the training error decreases to zero. This time, there is more randomness and the model requires almost as many parameters as there are data points to do memorize the data. The test error does show the typical bias-variance trade-off as we increase the capacity to the point where the training data is fitted exactly. However, then it does something unexpected; it starts to decrease again. Indeed, if we add enough capacity, the test loss reduces to below the minimal level that we achieved in the first part of the curve.

This phenomenon is known as *double descent*. For some datasets like MNIST it is present with the original data (figure 8.10c). For others, like MNIST1D and CIFAR (figure 8.10d), it emerges or becomes more prominent when we add noise to the labels. The first part of the curve is sometimes referred to as the *classical* or *under-parameterized regime* and the second part as the *modern* or *over-parameterized regime*. The part in the middle where the loss increases is sometimes termed the *critical regime*.

8.4.1 Explanation

The discovery of double descent is recent, unexpected, and somewhat puzzling. What we are seeing results from an interaction of two phenomena. First, the test performance becomes temporarily worse when the model has just enough capacity to memorize the data. Second, the test performance continues to improve with capacity even after the training performance is perfect. The first phenomenon is exactly as predicted by the bias-variance trade-off. The second phenomenon is more confusing; it's not clear why performance should be better in the over-parameterized regime given that there are not even enough training data points to uniquely constrain the model parameters here.

To help understand why the performance continues to improve as we add more parameters, note that once the model has enough capacity to drive the training loss to near zero, the model fits the training data almost perfectly. This implies that as we add further capacity we are not fitting the training data any better; consequently, any change must be occurring *between* the training points. The tendency of a model to prioritize one solution over another as it extrapolates between data points is known as its *inductive bias*.

Problems 8.4-8.5

Model behavior between data points is particularly important because in high-dimensional space, the training data are extremely sparse. The MNIST-1D dataset has 40 dimensions and we trained with 10,000 examples. If this seems like plenty of data, consider what would happen if we were to quantize each dimension of the input into 10 bins. There would be 10^{40} bins in total, and these are constrained by only 10^5 examples. Even with this coarse quantization, there will only be one data point in every 10^{35} bins! The tendency of the volume of high-dimensional space to overwhelm the number of training points is termed the *curse of dimensionality*.

The implication is that perhaps real problems in high dimensions look more like figure 8.11a. There are small regions of the input space where we observe data, but large gaps between them. The putative explanation for double descent is that as we

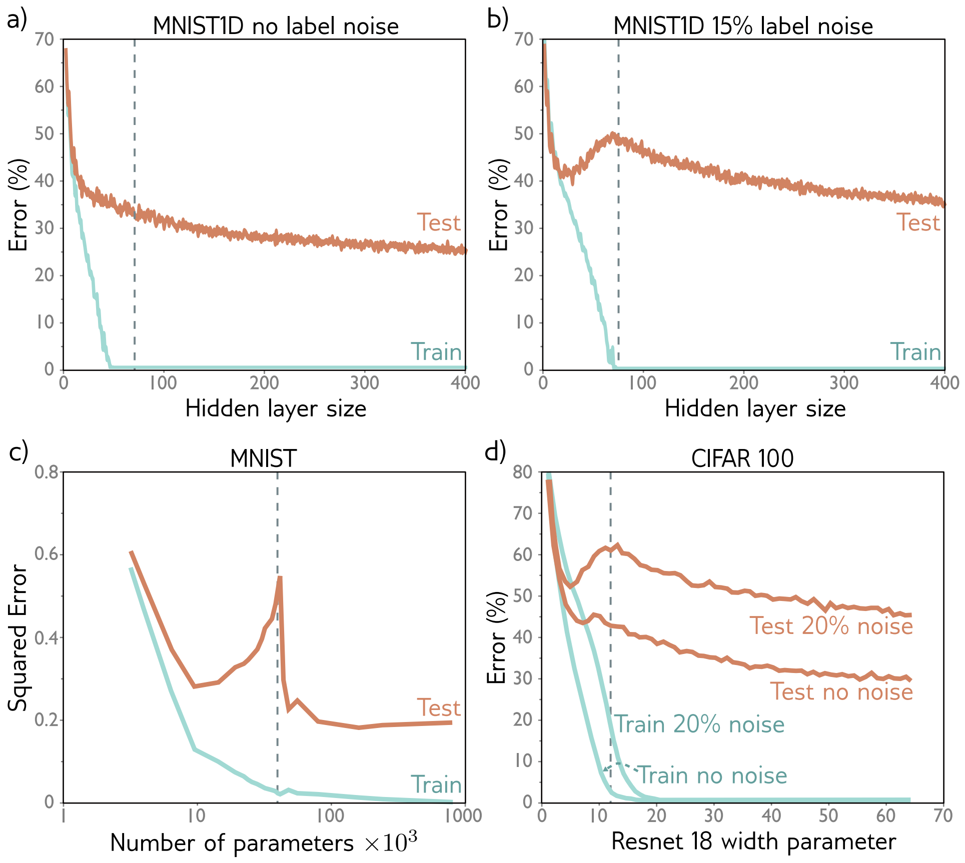


Figure 8.10 Double descent. a) Training and test loss on MNIST1D for a two hidden layer network as we increase the number of hidden units in each layer (and hence parameters). The training loss decreases to zero as the number of parameters approaches the number of training examples (vertical dashed line). The test error does not show the expected bias-variance trade-off but continues to decrease even after the model has memorized the dataset. b) The same experiment is repeated with noisier training data. Again, the training error reduces to zero, although it now takes almost as many parameters as there are training points to memorize the dataset. The test error shows the predicted bias/variance trade-off; it decreases as the capacity increases, but then increases again as we near the point where the training data is exactly memorized. However, it then subsequently decreases again and ultimately reaches a better performance level. This is known as double descent. Depending on the loss, the model, and the amount of noise in the data, the double descent pattern can be seen to a greater or lesser effect across many datasets. c) Results on MNIST dataset (without label noise) with shallow neural network from Belkin *et al.* (2019). d) Results on CIFAR 100 dataset with ResNet18 network (see chapter 11) from Nakkiran *et al.* (2019). See original papers for experimental details.

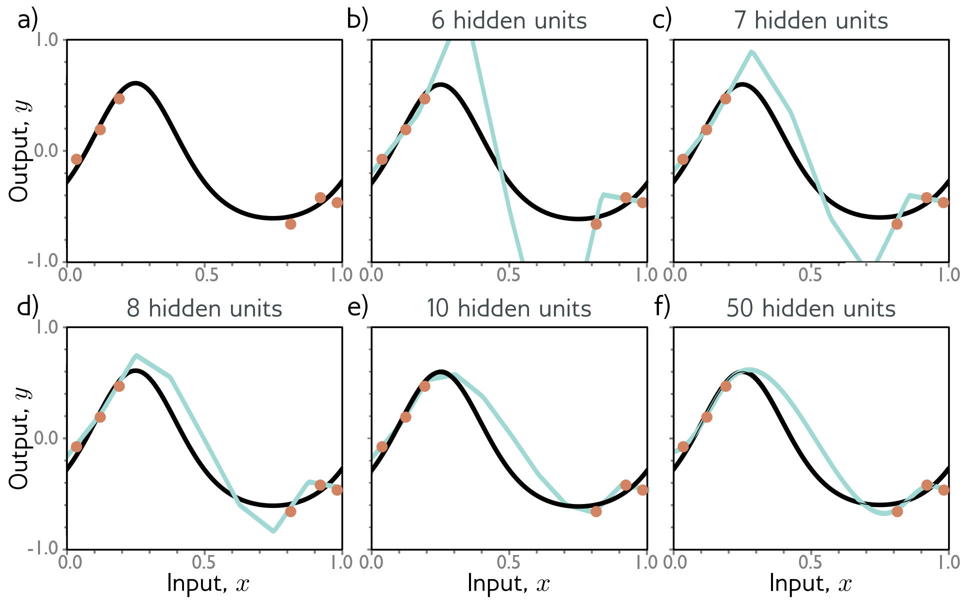


Figure 8.11 Increasing capacity allows smoother interpolation between sparse data points. a) Consider this situation where the training data (orange circles) are sparse and there is a large region in the center with no data examples to help the model fit the true function (black curve). b) If we fit a model that has just enough capacity to fit the training data (cyan curve), then it will have to contort itself to pass through the training data, and the output predictions will not be smooth. c-f) However, as we add more hidden units, the model has the *ability* to interpolate between the points more smoothly (smoothest possible curve plotted in each case). However, unlike here, it is not obliged to.

add more capacity to the model, the predicted function interpolates between the nearest data points increasingly smoothly. In the complete absence of information about what happens between the training points, assuming smoothness is a sensible strategy, which will probably generalize reasonably to new data.

This argument is plausible. It's certainly true that as we add more capacity to the model, it will have the capability to create smoother functions. Figures 8.11b-f show the smoothest possible functions that still pass through the data points as we increase the number of hidden units. When the capacity of the model is very close to the number of data examples (*i.e.*, ∼6 hidden units), it is forced to contort itself to fit them exactly, resulting in an erratic fit. This explains why the peak in the double descent curve is so pronounced. As we add more and more hidden units, the model has the ability to construct smoother and smoother functions that are likely to generalize to new data better.

However, this does not explain *why* over-parameterized models should produce

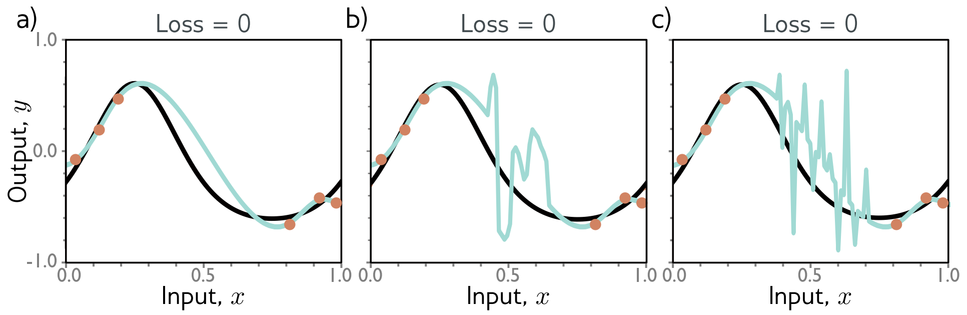


Figure 8.12 Regularization. a-c) Each of the three fitted curves passes through the data points exactly and so the training loss for each is zero. However, we might expect the smooth curve in panel (a) to generalize much better to new data than the erratic curves in panels (b) and (c). Any factor that biases a model towards a subset of the solutions with a similar training loss is known as a regularizer. It is thought that the initialization and/or fitting of neural networks has an implicit regularizing effect. Consequently, in the over-parameterized regime, more reasonable solutions such as that in panel (a) are encouraged.

smooth functions. Figure 8.12 shows three functions that can equally be created by the simplified model with 50 hidden units. In each case, the model fits the data exactly, and so the loss is zero. If the modern regime of double descent is explained by increasing smoothness, then what exactly is encouraging this smoothness?

There are two potential answers to this question. First, it's possible that the network initialization encourages smoothness and that during fitting the model never departs from this sub-domain of smooth functions. Second, it's possible that the training algorithm somehow 'prefers' to converge to smooth functions. Any factor that biases a solution towards a subset of equivalent solutions is known as a *regularizer*, and so one possibility is that the training algorithm acts as an implicit regularizer. We return to this topic in chapter 9.

8.5 Choosing hyperparameters

In the previous section, we discussed how the test performance changes as a function of the model capacity. Unfortunately, if we are in the classical regime, we don't have access to either the bias (which requires knowledge of the true underlying function) or the variance (which requires multiple independently sampled datasets to estimate). If we are in the modern regime, there is no way to tell how much capacity we need to add to improve the test loss. This raises the question of exactly how we should choose the capacity in practice.

For a deep network, the capacity of the model depends on the number of hidden layers, and the number of hidden units in each layer as well as other aspects of architecture that we have yet to introduce. Furthermore, we might expect that the choice of learning algorithms will also affect the final test performance. These elements are collectively termed the hyperparameters. The process of finding the best hyperparameters is known as *hyperparameter search*.

Hyperparameters are typically chosen empirically; we train many models with different hyperparameters on the same training set. We measure the performance for each and choose the model with the best result. However, we do not measure the performance on the test set; this would admit the possibility that the choice of hyperparameters just happens to work well for the test set but will not generalize to further data. Instead, we introduce a third set of data known as a *validation set*. For every choice of hyperparameters, we train the associated model using the training set and evaluate the performance on the validation set. Finally, we select the model that worked best on the validation set and evaluate its performance on the test set. In principle, this should give a good estimate of the true performance.

The hyperparameter space is generally smaller than the parameter space, but still too large to try every combination of hyperparameters exhaustively. Unfortunately, many hyperparameters are discrete (*e.g.*, the number of hidden layers), and others may be conditional on one another (*e.g.*, we only need to specify the number of hidden units in the tenth hidden layer if there are ten or more layers). Hence, we cannot rely on gradient descent techniques like those used for learning the model parameters. Hyperparameter optimization algorithms intelligently sample the space of hyperparameters, contingent on previous results. This procedure can be extremely computationally expensive since an entire model must be trained and the validation performance measured for each combination of hyperparameters.

8.6 Summary

To measure performance, we use a separate test set. The degree to which performance is maintained on this test set is known as generalization. Test errors can be explained by three factors: noise, bias, and variance. These combine additively in regression problems with least squares losses. Adding training data decreases the variance. When the model capacity is less than the number of training examples, increasing the capacity decreases bias, but increases variance. This is known as the bias-variance trade-off, and there is a capacity where the trade-off is optimal.

However, this is balanced against a tendency for performance to improve with capacity, even when the parameter exceeds the training examples. Together, these two phenomena create the double descent curve. It is thought that the model interpolates more smoothly between the training data points in the over-parameterized 'modern regime' although it is not clear what drives this. To choose the capacity and other model and training algorithm hyperparameters, we fit multiple models and evaluate their performance using a separate validation set.

Notes

Bias-variance trade-off: We showed that the test error for regression problems with least squares loss decomposes into the sum of noise, bias, and variance terms. These factors are all present for models with other losses, but their interaction is typically more complicated (Friedman 1997; Domingos 2000). For classification problems there are some counter-intuitive predictions; for example, if the model is biased towards selecting the wrong class in a region of the input space, then increasing the variance can improve the classification rate as this pushes some of the predictions over the threshold to be classified correctly.

Cross-validation: We saw that it is typical to divide the data into three parts: training data (which is used to choose the parameters), validation data (which is used to choose the hyperparameters), and test data (which is used to estimate the final performance). This approach is known as *cross-validation*. However, this division may cause problems in situations where the total number of data examples is limited; as we saw earlier in this chapter, if the number of training examples is comparable to the model capacity, then the variance will be large.

One way to reduce this problem is to use *k-fold cross-validation*. The training and validation data are partitioned into K disjoint subsets. For example, we might divide these data into five parts. We train with four and validate with the fifth for each of the five permutations and choose the hyperparameters based on the average validation performance. The final test performance is assessed by using the average of the predictions from the five models with the best hyperparameters on a completely different test set. There are many variations of this idea, but all share the general goal of using a larger proportion of the data to train the model and hence reduce variance.

Capacity: We have used the term *capacity* informally to mean the number of parameters or hidden units in the model (and hence indirectly, the ability of the model to fit functions of increasing complexity). The *representational capacity* of a model describes the space of possible functions it can construct when we consider all possible parameter values. When we take into account the fact that an optimization algorithm may not be able to reach all of these solutions, what is left is the *effective capacity*.

The Vapnik-Chervonenkis (VC) dimension (Vapnik & Chervonenkis 1971) is a more formal measure of capacity. It is the largest number of training examples that a binary classifier can label arbitrarily. Bartlett *et al.* (2017b) derive upper and lower bounds for the VC-dimension in terms of the number of layers and weights. An alternative measure of capacity is the Rademacher complexity, which is the expected empirical performance of a classification model (with optimal parameters) for data with random labels. Neyshabur *et al.* (2017a) derive a lower bound on the generalization error in terms of the Rademacher complexity.

Double descent: The term 'double descent' was coined by Belkin *et al.* (2019) who demonstrated that the test error decreases again in the over-parameterized regime for two-layer neural networks and random features. They also claimed that this occurs in decision trees although Buschjäger & Morik (2021) subsequently provided evidence to the contrary. Nakkiran *et al.* (2019) show that double descent occurs for various modern datasets (CIFAR-10, CIFAR-100, IWSLT'14 de-en), model architectures (CNNs, ResNets, transformers), and optimizers (SGD, Adam). The double descent phenomenon is more pronounced when noise is added to the target labels citenakkiran2019deep and also when some regularization techniques are used (Ishida *et al.* 2020).

Nakkiran *et al.* (2019) also provide empirical evidence that test performance is dependent on *effective model capacity*, which is the largest number of samples for which a given model and training method can achieve zero training error. This is the point at which the model starts to devote its efforts to interpolating smoothly. As such, the test performance depends not just on the model, but also the training algorithm and length of training. They observe the same pattern when they study a model with fixed capacity and increase the number of training iterations. They term this *epoch-wise double descent*.

Double descent makes the rather strange prediction that adding training data can sometimes make test performance worse. Consider an over-parameterized model that is in the second descending part of the curve. If we increase the amount of training data so that it matches the model capacity, we will now be in the critical region of the new test error curve and the test loss may increase.

Bubeck & Sellke (2021) prove that overparameterization is necessary if we want to interpolate data smoothly in high dimensions. They demonstrate that there is a trade-off between the number of parameters and the Lipschitz constant of a model (the fastest the output can change for a small change in the input). A review of the theory of overparameterized machine learning can be found in Dar *et al.* (2021).

Curse of dimensionality: As dimensionality increases, the volume of space grows so fast that the amount of data needed to densely sample it increases exponentially. This phenomenon is known as the curse of dimensionality. In general, high-dimensional space has many unexpected properties, and caution should be used when trying to reason about it based on low-dimensional examples. This book visualizes many aspects of deep learning in one or two dimensions, but these visualizations should be treated with healthy skepticism.

Problems 8.6-8.9

The following are some of the surprising properties of high-dimensional space: (i) Two randomly sampled data points from a standard normal distribution are very close to orthogonal to one another with high likelihood. (ii) The distance from the origin of samples from a standard normal distribution is roughly constant. (iii) Most of a volume of a high-dimensional sphere (hypersphere) is adjacent to its surface (a common metaphor is that most of the volume of a high-dimensional orange is in the peel, not in the pulp). (iv) If we place a unit diameter hypersphere inside a hypercube with unit length sides, then the hypersphere takes up a decreasing proportion of the volume of the cube as the dimension increases. Since the volume of the cube is fixed at size one, this implies that the volume of a high-dimensional hypersphere becomes close to zero. (v) If we generate a set of random points in a high-dimensional hypercube, then the ratio of the Euclidean distance between the nearest and furthest points becomes close to one. For further information, consult Beyer *et al.* (1999) and Aggarwal *et al.* (2001).

Real-world performance: In this chapter, we argued that model performance can be evaluated by using a held-out test set. However, this is not necessarily indicative of real-world performance if the statistics of the test set do not match the statistics of data in the real world. Moreover, the statistics of real-world data may change over time, causing the model to become increasingly stale and performance to decrease. This is known as *data drift* and means that deployed models must be carefully monitored.

There are three main reasons why real-world performance may be worse than the test performance implies. First, the statistics of the input data $\mathbf{x}$ may change; we may then be observing parts of the function that were sparsely sampled or not sampled at all during training. This is known as *covariate shift*. Second, the statistics of the output data $\mathbf{y}$ may change; if some values are very rare during training, then the model may learn not to predict these in ambiguous situations and will make mistakes if they are more common in the real world. This is known as *prior shift*. Third, the relationship between input and output may change. This is known as *concept shift*. These issues are discussed in Moreno-Torres *et al.* (2012).

Hyperparameter search: Finding the best hyperparameters is a challenging optimization task. Testing a single configuration of hyperparameters is expensive; we have to train an entire model and measure its performance. We have no easy way to access the derivatives (*i.e.*, how performance changes when we make a small change to a hyperparameter) and anyway many of the hyperparameters are discrete, so we cannot use gradient descent methods. There are multiple local minima and no way to tell if we are close to the global minimum. The noise level is high since each training/validation cycle uses a stochastic training algorithm, and we don't expect to get the same results if we train a model twice with the same hyperparameters. Finally, some variables are conditional and only exist if others are set. For example, the number of neurons in the third hidden layer of a network is only relevant if we have at least three hidden layers.

A very simple approach is just to randomly sample the space (Bergstra & Bengio 2012). However, for continuous variables, a more intelligent method is to build a model of the underlying hyper-parameter/performance function, and the uncertainty in this function. Then we can exploit this to test where the uncertainty is great (explore the space), or home in on regions where performance looks promising (exploit previous knowledge). Bayesian optimization is a framework based on Gaussian processes that does just this and its application to hyperparameter search is described in Snoek *et al.* (2012). A roughly equivalent model for describing the uncertainty in results due to discrete variables, is the Beta-Bernoulli bandit (see Lattimore & Szepesvári 2020).

The sequential model-based configuration (SMAC) algorithm (Hutter *et al.* 2011) can cope with continuous, discrete, and conditional parameters. The basic approach is to use a random forest to model the objective function where the mean of the tree predictions is the best guess about the objective function and their variance represents the uncertainty. A completely different approach that can also cope with combinations of continuous, discrete, and conditional parameters is Tree-Parzen Estimators (Bergstra *et al.* 2011). The previous methods modeled the probability of the model performance given the hyperparameters. In contrast, the Tree-Parzen estimator models the probability of the hyperparameters given the model performance.

Hyperband (Li *et al.* 2016b) is a multi-armed bandit strategy for hyperparameter optimization. It assumes that there are computationally cheap but approximate ways to measure performance (*e.g.*, by not training to completion), and that these can be associated with a budget (*e.g.*, by training for a fixed number of iterations). A number of random configurations are sampled and run until the budget is used up. Then the best fraction η of runs is kept and the budget is multiplied by $1/\eta$. This is repeated until the maximum budget is reached. This approach has the advantage of efficiency; for bad configurations, it does not need to run the experiment to the end. However, each sample is just chosen randomly which is inefficient. The BOHB algorithm (Falkner *et al.* 2018) combines the efficiency of Hyperband with the more sensible choice of parameters from Tree Parzen estimators to construct an even better hyperparameter optimization method.

Problems

Problem 8.1 Will the multi-class cross-entropy training loss in figure 8.2 ever reach zero? Explain your reasoning.

Problem 8.2 What values should we choose for the three weights and the bias in the first layer of the model in figure 8.4a so that the responses at the hidden units are as depicted in figures 8.4b-d?

Problem 8.3 Given a training dataset consisting of I input/output pairs $\{x_i, y_i\}$, show how the parameters $\{\beta, \omega_1, \omega_2, \omega_3\}$ can be found in closed form for the model in figure 8.4a using the least squares loss function.

Problem 8.4 Consider the curve in figure 8.10b at the point where we train a model with a hidden layer of size 200, which would have 50,410 parameters. What do you predict will happen to the training and test performance if we increase the number of training examples from 10,000 to 50,410?

Problem 8.5 Consider the case where the model capacity exceeds the number of training data points, and the model is flexible enough to reduce the training loss to zero. What are the implications of this for fitting a heteroscedastic model? Propose a method to resolve any problems that you identify.

Problem 8.6 Prove that the angle between two random samples from a 1000 dimensional standard Gaussian distribution is close to zero with high probability.

Problem 8.7 The volume of a hypersphere with radius r in D dimensions is:

$$\mathrm{Vol}[r] = \frac{r^D \pi^{D/2}}{\Gamma[D/2+1]}, \tag{8.8}$$

where $\Gamma[\bullet]$ is the Gamma function. Show that the volume of a hypersphere of diameter one (radius $r = 0.5$) becomes zero as the dimension increases.

Problem 8.8 Consider a hypersphere of radius $r = 1$. Calculate the proportion of the total volume that is in the 1% of the radius that is closest to the surface of the hypersphere as a function of the dimension.

Problem 8.9 Figure 8.13c shows the distribution of distances of samples of a standard normal distribution as the dimension increases. Empirically verify this finding by sampling from the standard normal distributions in 25, 100, and 500 dimensions and plotting a histogram of the distances from the center. What closed-form probability distribution describes these distances?

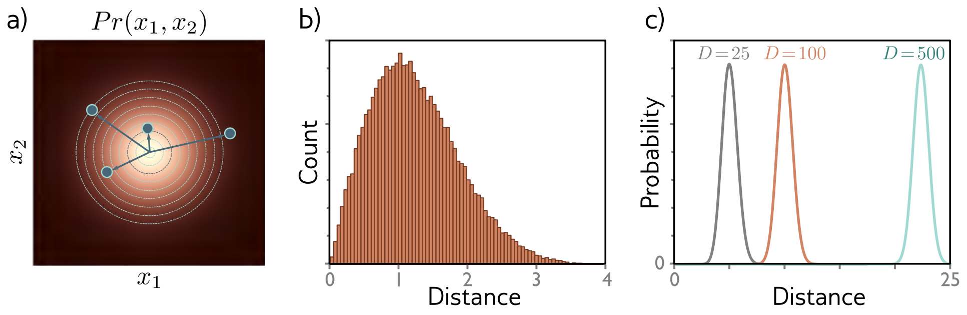


Figure 8.13 Typical sets. a) Standard normal distribution in two dimensions. Circles indicate four samples from this distribution. As the distance from the center increases the probability decreases, but the volume of space at that radius (*i.e.*, the area between adjacent evenly spaced circles) increases. b) These factors trade off so that the histogram of distances of samples from the center has a pronounced peak. c) As we increase the dimensionality of the normal distribution, this effect becomes more extreme and the probability of observing a sample with a small distance from the mean becomes vanishingly small. Although the most likely point is at the mean of the distribution, the *typical samples* are found in a relatively narrow shell.

Chapter 9

Regularization

Chapter 8 described how to measure model performance and identified that there can be a large performance gap between the training and test data. Possible reasons for this discrepancy include: (i) the model describes statistical peculiarities of the training data that are not representative of the true mapping from input to output (overfitting), and (ii) the model is unconstrained in areas where there are no training examples, and this leads to unpredictable behavior.

This chapter discusses *regularization* techniques. These are a family of methods that reduce the generalization gap between training and test performance. Strictly speaking, regularization involves adding explicit terms to the loss function that favor certain parts of parameter space. However, in machine learning, this term is often used to refer to any strategy that improves generalization.

We start by considering regularization in its strictest sense. Then we'll consider how the stochastic gradient descent algorithm itself favors certain solutions. This is known as implicit regularization. Following this, we'll consider a set of heuristic methods that improve test performance. These include early stopping, ensembling, dropout, label smoothing, and transfer learning.

9.1 Explicit regularization

Consider fitting a model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ using a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek the minimum of the loss function $L[\boldsymbol{\phi}]$:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\mathrm{L}[\boldsymbol{\phi}]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I} l_i[\mathbf{x}_i, \mathbf{y}_i]\right], \end{aligned} \tag{9.1}$$

where the individual terms $l_i[\mathbf{x}_i, \mathbf{y}_i]$ measure the mismatch between the network

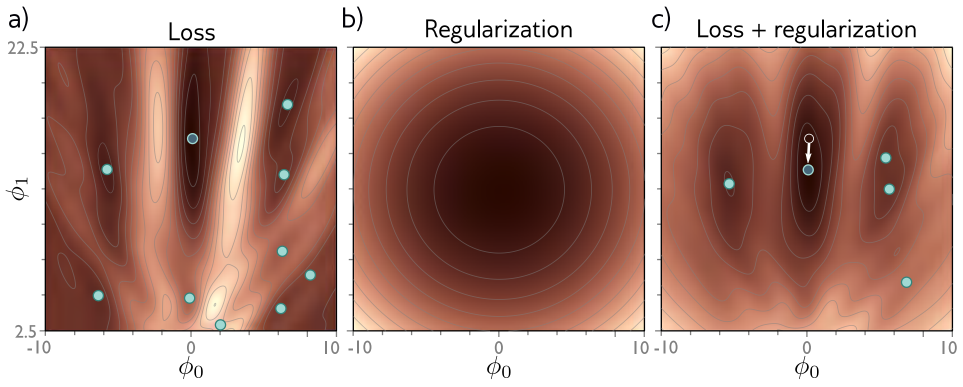


Figure 9.1 Explicit regularization. a) Loss function for Gabor model (see section 6.1.2). Cyan circles represent local minima, green circle represents the global minimum. b) The regularization term favors parameters close to the center of the plot by adding an increasing penalty as we move away from this point. c) The final loss function is the sum of the original loss function plus the regularization term. This surface has fewer local minima, and the global minimum has moved to a different position (arrow shows change).

predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and output targets $\mathbf{y}_i$ for each training pair. To bias this minimization towards certain solutions, we add an extra term:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} l_i[\mathbf{x}_i, \mathbf{y}_i] + \lambda \cdot \mathrm{g}[\boldsymbol{\phi}] \right], \tag{9.2}$$

where $\mathrm{g}[\boldsymbol{\phi}]$ is a function that returns a scalar that takes a larger value when the parameters are less preferred. The term λ is a positive scalar that controls the relative contribution of the original loss function and the regularization term. The minima of the regularized loss function usually differ from those in the original, and so the training procedure converges to different parameter values (figure 9.1).

9.1.1 Probabilistic interpretation

Regularization can be viewed from a probabilistic perspective. Section 5.1 shows how loss functions are constructed from the maximum likelihood criterion:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i | \mathbf{x}_i, \boldsymbol{\phi}) \right]. \tag{9.3}$$

The regularization term can be considered as applying a prior $Pr(\boldsymbol{\phi})$ that represents knowledge about the parameters before we observe the data:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\mathrm{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i,\boldsymbol{\phi})Pr(\boldsymbol{\phi})\right]. \tag{9.4}$$

Moving back to the negative log-likelihood loss function by taking the log and multiplying by minus one, we see that $\lambda \cdot \mathrm{g}[\boldsymbol{\phi}] = -\log[Pr(\boldsymbol{\phi})]$.

9.1.2 L2 regularization

This discussion has sidestepped the question of exactly which solutions the regularization term should penalize (or equivalently that the prior should favor). Since neural networks are used in an extremely broad range of applications, these can only be very generic preferences. The most used regularization term is the *L2 norm*, which penalizes the sum of the squares of the parameter values:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\left[\mathrm{L}[\boldsymbol{\phi},\{\mathbf{x}_i,\mathbf{y}_i\}] + \lambda\sum_{k}\phi_k^2\right], \tag{9.5}$$

where k indexes the parameters. This is also referred to as *Tikhonov regularization*, *Frobenius norm regularization*, or *ridge regression*. Problems 9.1-9.2

For neural networks, L2 regularization term is usually applied to the weights but not the biases and hence this is referred to as a *weight decay* term. The effect is to encourage the weights to be smaller and hence the output function to be smoother. To see this, consider that the output prediction is a weighted sum of the activations at the last hidden layer. If the weights have a smaller magnitude, then the output will vary less. The same logic applies to the computation of the pre-activations at the last hidden layer, and so on progressing backward through the network. In the limit, if we forced all the weights to be zero, the network would produce a constant output determined by the bias parameters.

Figure 9.2 shows the effect of fitting the simplified network from figure 8.4 with weight decay and different values of the regularization coefficient λ. When λ is small, it has little effect. However, as λ increases, the fit to the data becomes less accurate, and the function becomes smoother. This might improve the test performance for two reasons:

- If the network is overfitting, then adding the regularization term means that the network must trade off slavish adherence to the data against the desire to be smooth. One way to think about this is that the error due to variance reduces (the model no longer needs to pass through every data point), at the cost of increased bias (the model can only describe smooth functions).
- When the network is over-parameterized, some of the extra model capacity describes areas where there were no training data. Here, the regularization term will favor functions that smoothly interpolate between the nearby points. This is reasonable behavior in the absence of knowledge about the true function.

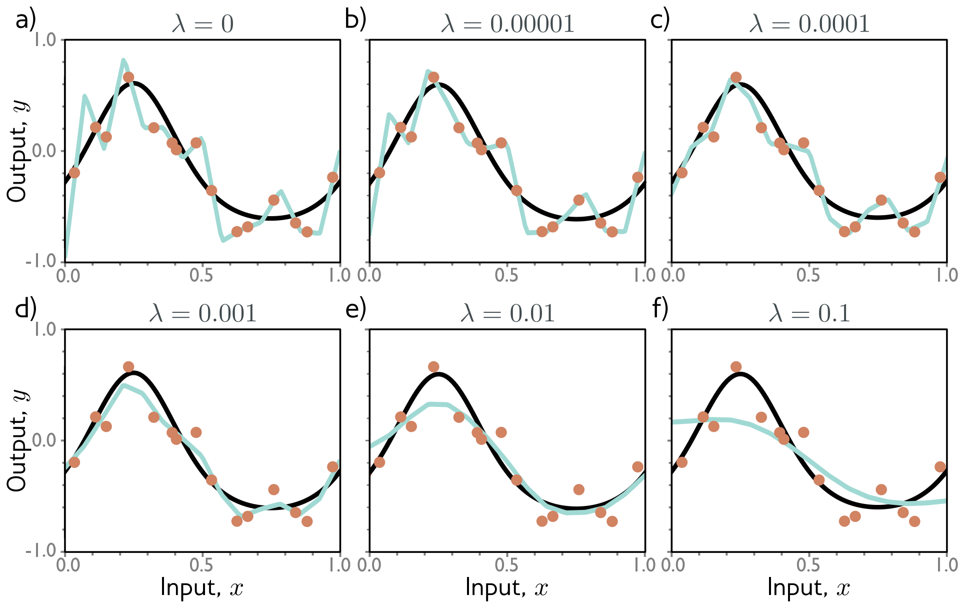


Figure 9.2 L2 regularization in simplified neural network (see figure 8.4). a-f) Fitted functions as we increase the regularization coefficient λ. In each panel, the black curve is the true function, the orange circles are the noisy training data, and the cyan curve is the fitted function. For small values of λ (panels a-b), the fitted function passes exactly through the data points. For intermediate values (panels c-d), the function becomes smoother and more similar to the ground truth curve. For large values (panels e-f), the fitted function is forced to be smoother than the ground truth and so the fit becomes worse.

9.2 Implicit regularization

An intriguing recent finding is that neither gradient descent nor stochastic gradient descent descend neutrally to the minimum of the loss function; each exhibits a preference for some solutions over others. This is known as *implicit regularization*.

9.2.1 Implicit regularization for gradient descent

Consider a continuous version of gradient descent where the step size is infinitesimal. The change in parameters $\boldsymbol{\phi}$ will be governed by the differential equation:

$$\frac{\partial \boldsymbol{\phi}}{\partial t} = -\frac{\partial L}{\partial \boldsymbol{\phi}}. \tag{9.6}$$

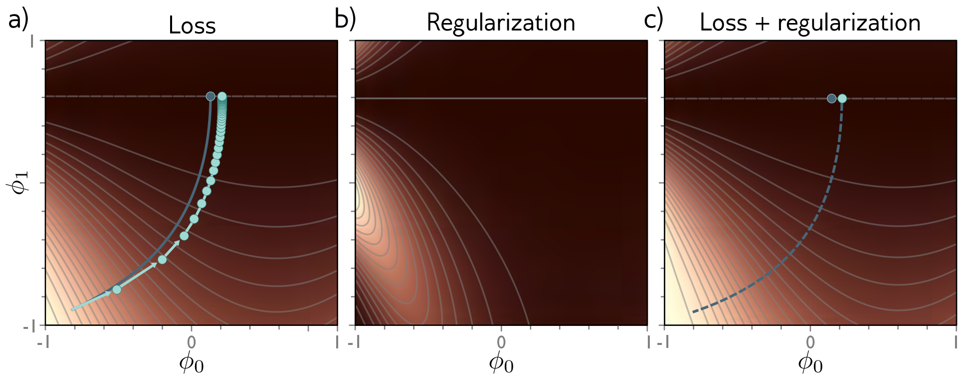


Figure 9.3 Implicit regularization in gradient descent. a) Loss function with family of global minima on horizontal line $\phi_1 = 0.61$. Dashed blue line shows continuous gradient descent path starting in bottom left corner. Solid cyan line shows discrete gradient descent with step size 0.1 (first few steps shown explicitly as arrows). The finite step size causes the paths to diverge and to reach a different final position. b) This disparity can be approximated by adding a regularization term that penalizes the squared gradient magnitude to the continuous gradient descent loss function. c) After adding this term, the continuous gradient descent path converges to the same place that the discrete one did on the original function.

Gradient descent approximates this process with a series of discrete steps:

$$\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}_t - \alpha \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{9.7}$$

where α is the step size. The discretization means that the algorithm deviates from the continuous path (figure 9.3).

This deviation can be understood by deriving a modified loss term $\tilde{L}$ for the continuous case that arrives at the same place as the discretized version on the original loss L. It can be shown (see end of chapter) that this modified loss is:

$$\tilde{\mathrm{L}}_{GD}[\boldsymbol{\phi}] = \mathrm{L}[\boldsymbol{\phi}] + \frac{\alpha}{4} \left\| \frac{\partial L}{\partial \boldsymbol{\phi}} \right\|^2. \tag{9.8}$$

In other words, the trajectory of the discrete version will be repelled from places where the norm of the gradient is very large, and the surface is steep. This does not affect the position of the minima (where the gradients are zero) but does modify the effective loss function elsewhere and so causes the solution to take a different trajectory and potentially converge to a different minimum. Implicit regularization due to gradient descent is probably responsible for the observation that full batch gradient descent generalizes better with larger step sizes (figure 9.5a).

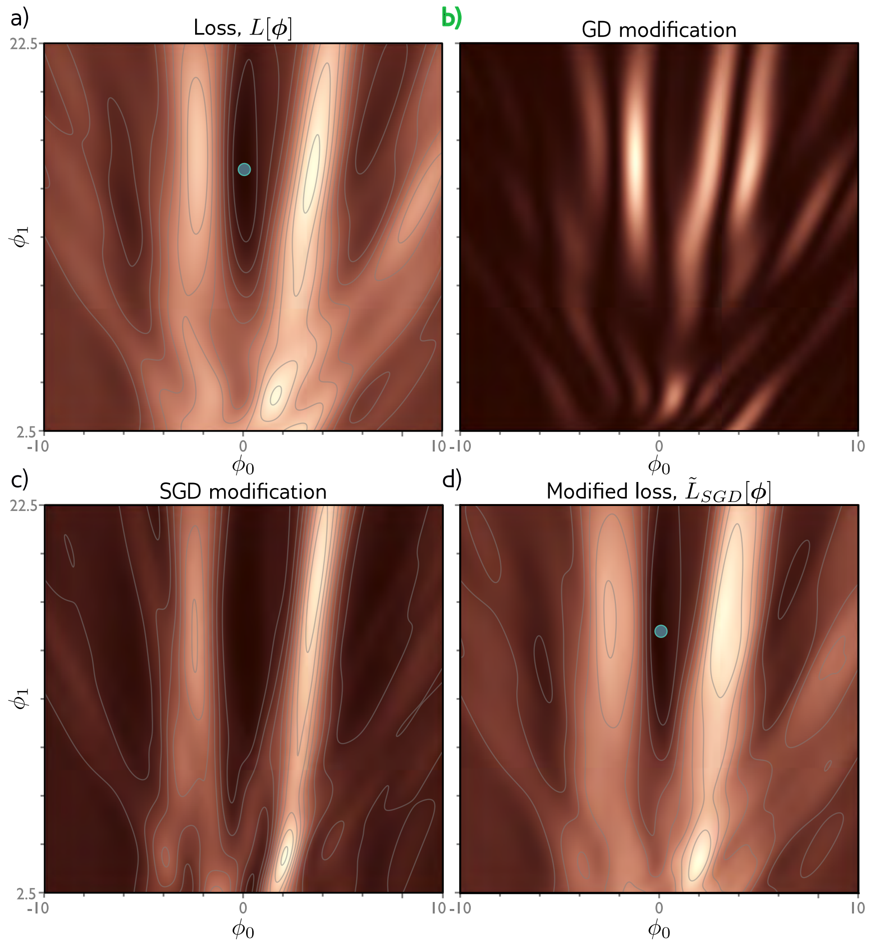


Figure 9.4 Implicit regularization for stochastic gradient descent. a) Original loss function for Gabor model (section 6.1.2). b) Implicit regularization term from gradient descent penalizes the squared gradient magnitude. c) Additional implicit regularization from stochastic gradient descent penalizes the variance of the batch gradients. d) Modified loss function (sum of original loss plus two implicit regularization components).

9.2.2 Implicit regularization in stochastic gradient descent

A similar analysis can be applied to stochastic gradient descent. Now we seek a modified loss function such that the continuous version reaches the same place as the average of the possible random SGD updates. This can be shown to be:

$$
\begin{aligned}
\tilde{\mathrm{L}}_{SGD}[\boldsymbol{\phi}] &= \tilde{\mathrm{L}}_{GD}[\boldsymbol{\phi}] + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 \\
&= \mathrm{L}[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2. \qquad (9.9)
\end{aligned}
$$

Here, L_b is the loss for the b^{th} of the B batches in an epoch and both L and L_b now represent the means of the I individual losses in the full dataset and the $|\mathcal{B}|$ individual losses in the batch respectively:

$$
L = \frac{1}{I}\sum_{i=1}^{I} l_i[\mathbf{x}_i, y_i] \qquad \text{and} \qquad L_b = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_b} l_i[\mathbf{x}_i, y_i]. \qquad (9.10)
$$

Equation 9.9 reveals an extra regularization term, which corresponds to the variance of the gradients of the batch losses L_b. In other words, SGD implicitly favors places where the gradients are stable (where all the batches agree on the slope). Once more this modifies the trajectory of the optimization process (figure 9.4) but does not necessarily change the position of the global minimum; if the model is over-parameterized, then it may fit all the training data exactly and so all these gradient terms will all be zero.

SGD generalizes better than gradient descent and smaller batch sizes usually perform better (figure 9.5b). One possible explanation is that the inherent randomness allows the algorithm to reach different parts of the loss function. However, it's also possible that some or all of this performance increase is due to implicit regularization; this encourages solutions where all the data fits well (and so the batch variance is small) rather than solutions where some of the data fit extremely well and other data less well (perhaps with the same overall loss, but with larger batch variance). The former solutions are likely to generalize better.

9.3 Heuristics to improve performance

We've seen that regularization encourages the network to find a good solution by adding extra terms to the loss function. This can be done explicitly, but also occurs implicitly as an unintended (but seemingly helpful) byproduct of stochastic gradient descent. In this section, we discuss other heuristic methods that have been used to improve generalization.

9.3.1 Early stopping

As the name suggests, *early stopping* refers to the practice of stopping the training procedure before it has fully converged. This can reduce overfitting because the

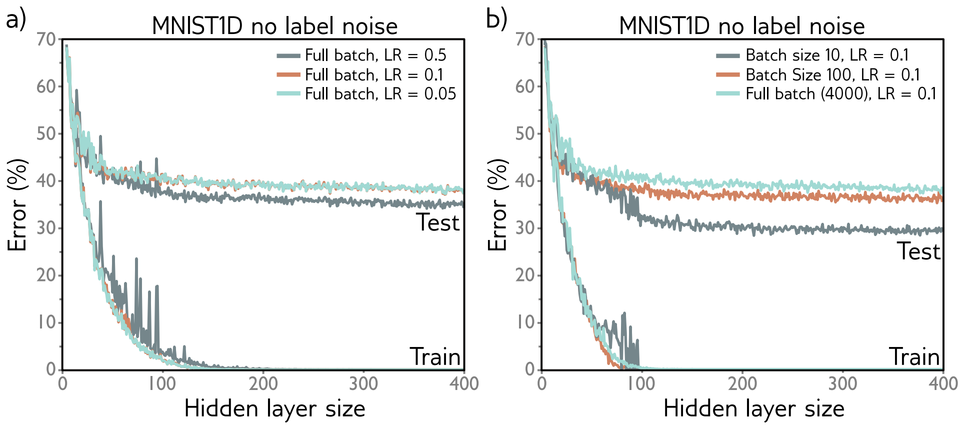


Figure 9.5 Effect of learning rate and batch size for 4000 training examples and 4000 test examples of 1D MNIST data (see figure 8.1) for a neural network with two hidden layers. a) Performance is better for large learning rates than for intermediate or small ones (full batch gradient descent case shown). In each case, the number of iterations is 6000× the learning rate so each solution has the opportunity to move the same distance. b) Performance is superior for smaller batch sizes. In each case the number of iterations was chosen so that the training data were memorized at roughly the same model capacity.

model may have already captured the coarse shape of the underlying function, but not yet had time to overfit to the noise (figure 9.6).

One way of thinking about this is that since the weights are initialized to small values (see section 7.3), they simply don't have time to become large, and so early stopping has a similar effect to explicit L2 regularization. A different view is that early stopping reduces the effective model complexity. Hence, we move back down the bias/variance trade-off curve from the critical region and performance improves (see figures 8.9 and 8.10).

Early stopping has a single associated hyperparameter, which is the number of optimization steps after which learning is terminated. As for other hyperparameters, this is chosen empirically using a validation set (section 8.5). However, for early stopping, this hyperparameter can be chosen without the need to train multiple models. The model is trained once, the performance on the validation set is monitored every T iterations and the associated models are stored. The stored model where the validation performance was best is selected.

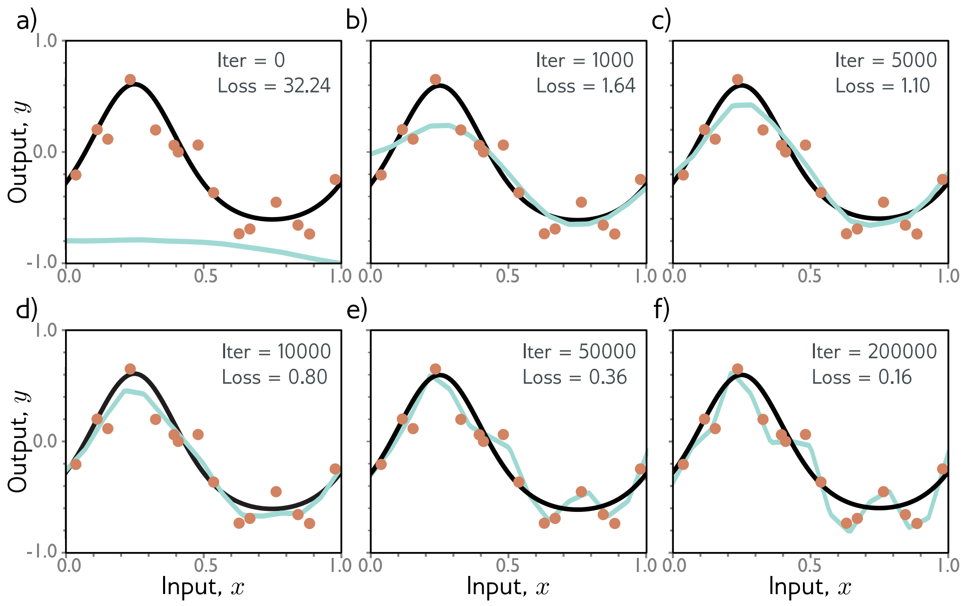


Figure 9.6 Early stopping. a) A simplified shallow neural network model with 14 linear regions (figure 8.4) is initialized randomly (cyan curve) and trained with SGD using a batch size of five and a learning rate of 0.05. b-d) As training proceeds, the function first captures the coarse structure of the true function (black curve), before e-f) overfitting to the noisy training data (orange points). Although the training loss continues to decrease throughout this process, the learned models in panels (c) and (d) are closest to the true underlying function and will generalize better to test data than those in panels (e) or (f).

9.3.2 Ensembling

A completely different approach to reducing the generalization gap between training and test data, is to build several models and average their predictions. A group of such models is known as an *ensemble*. This technique reliably improves test performance at the cost of training and storing multiple models and performing inference multiple times.

The models can be combined by taking the mean of the outputs (for regression problems) or the mean of the pre-softmax activations (for classification problems). The underlying assumption here is that the errors in the different models are independent and will cancel out. Alternatively, we can take the median of the outputs (for regression problems) or the most frequent predicted class (for classification problems) to make the predictions more robust.

One way to train different models is just to use different random initializations.

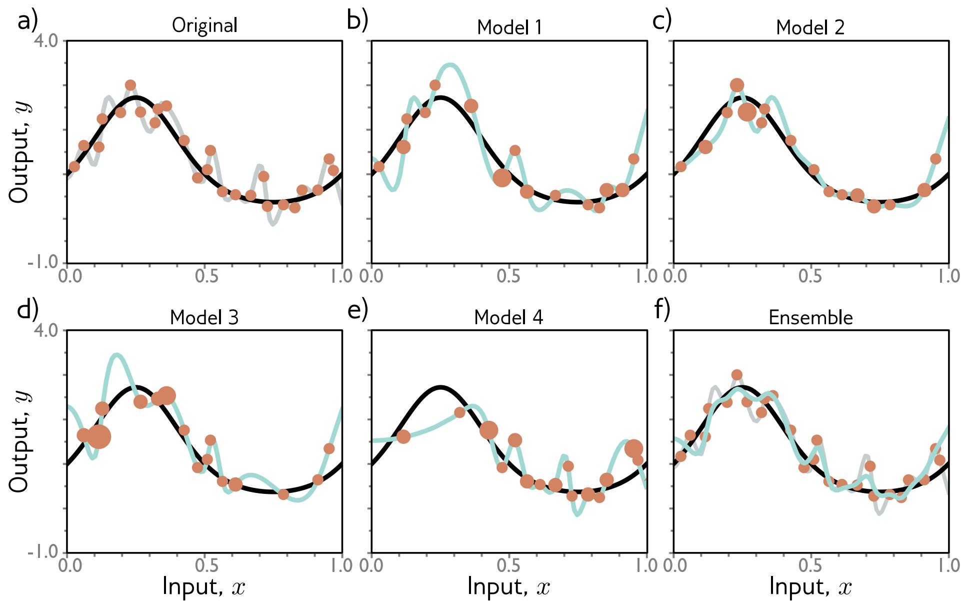


Figure 9.7 Ensemble methods. a) Fitting a single model (gray curve) to the entire dataset (orange points). b-e) An ensemble of four models is created by re-sampling the data with replacement (bagging) four times and fitting a model to each (size of orange point indicates number of times the data point was re-sampled). f) When we average these predictions (cyan curve), the result is smoother than the result from panel (a) for the full dataset (gray curve) and will probably generalize better.

This may particularly help in regions of the input space that are far from the training data and where the fitted function is relatively unconstrained. Different models may produce very different predictions in these regions and so the average of several models may generalize better than any individual model.

A second approach is to generate several different datasets by re-sampling the training data with replacement and training a different model from each. This is known as *bootstrap aggregating* or *bagging* for short (figure 9.7). It has the effect of smoothing out the data; if a data point is not present in one training set, then the model will interpolate from nearby points; hence, if that point was an outlier, then the fitted function will be generally more moderate in this region. Other approaches include training models with different hyperparameters, or training completely different families of models.

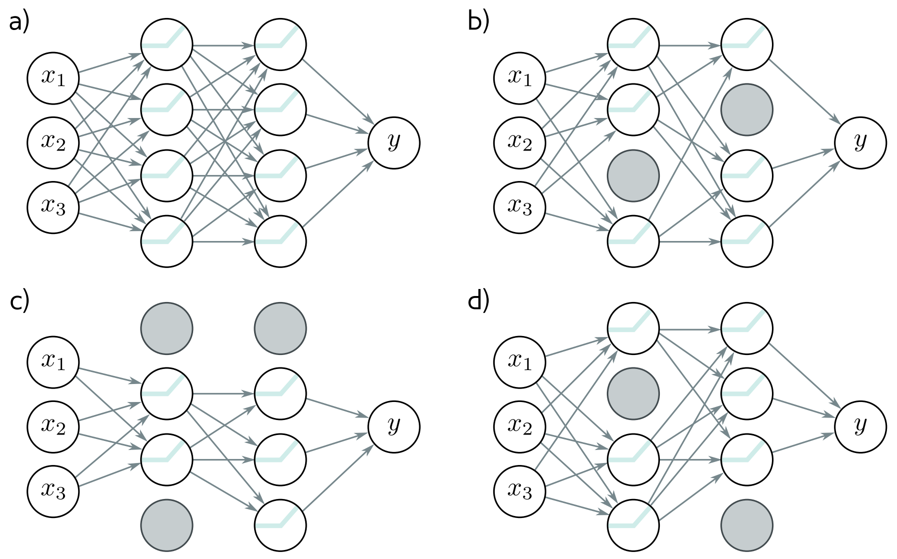


Figure 9.8 Dropout. a) Original network. b-d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes). The result is that the incoming and outgoing weights from these units have no effect and so we are training with a slightly different network each time.

9.3.3 Dropout

Dropout randomly clamps a subset (typically 50%) of hidden units to zero at each iteration of SGD (figure 9.8). This makes the network less dependent on any given hidden unit; this encourages the weights to have smaller magnitudes so that the change in the function due to the presence or absence of the hidden unit is reduced.

This technique has the positive benefit that it can eliminate kinks in the function that are far from the training data and so don't affect the loss. For example, consider three hidden units that become active sequentially as we move along the curve (figure 9.9a). The first hidden unit causes a large increase in the slope. A second hidden unit decreases the slope so that the function goes back down. Finally, the third unit cancels out this decrease and returns the subsequent curve to its original trajectory. These three units conspire to make an undesirable local change in the function. This will not change the training loss but is unlikely to generalize well.

When several units conspire to make an undesirable local change, eliminating one of them (as would happen in dropout) causes a huge change in the output function that is propagated to the half-space where that unit was active (figure 9.9b). A subsequent gradient descent step will attempt to compensate for the change that this induces, and over time such dependencies will be eliminated. The overall ef-

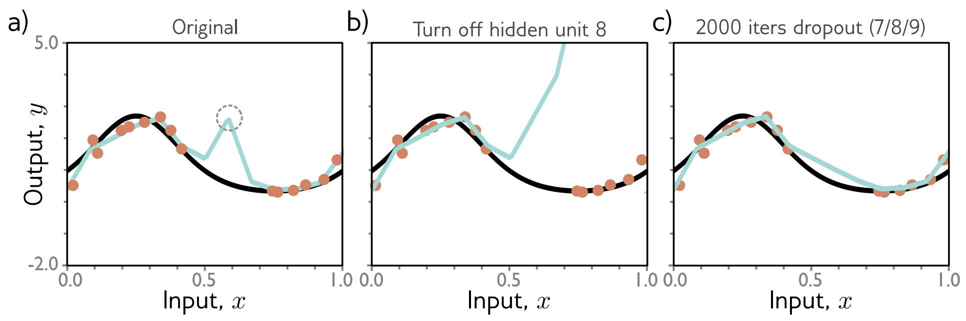


Figure 9.9 Dropout mechanism. a) There is an undesirable kink in the curve caused by a sequential increase in the slope, decrease in the slope (at circled joint), and then another increase to return the curve to its original trajectory. Here we are using full-batch gradient descent and the model already fits the data as well as possible, so further training will not remove this kink. b) Consider what happens if we remove the hidden unit that produced the circled joint in panel (a) as might happen using dropout. Without the decrease in the slope, the right-hand side of the function takes an upwards trajectory and a subsequent gradient descent step will aim to compensate for this change. c) Curve after 2000 iterations of (i) randomly removing one of the three hidden units that cause the kink, and (ii) performing a gradient descent step. Although the kink does not affect the loss, it is nonetheless removed by this approximation of the dropout mechanism.

fect is that large unnecessary changes between training data points are gradually removed even though they contribute nothing to the loss (figure 9.9).

At test time, we can run the network as normal with all the hidden units active; however, the network now has more hidden units than it was trained with at any given iteration, so to compensate for this it is typical to multiply the weights in the network by the dropout probability. This is known as the *weight scaling inference rule*. A different approach to inference is to use *Monte Carlo dropout*, in which we run the network multiple times with different units clamped to zero exactly as in training and combine the results. This is closely related to ensembling in that every random version of the network is a different model; however, here we do not have to train or store multiple networks.

9.3.4 Applying noise

Dropout can be interpreted as applying multiplicative Bernoulli noise to the activations of the network. This leads to the idea of applying noise to other parts of the network during training, to make the final model more robust.

Problem 9.3

One possibility is to add random noise to the input data; this has the effect of smoothing out the learned function (figure 9.10). For regression problems, it can be

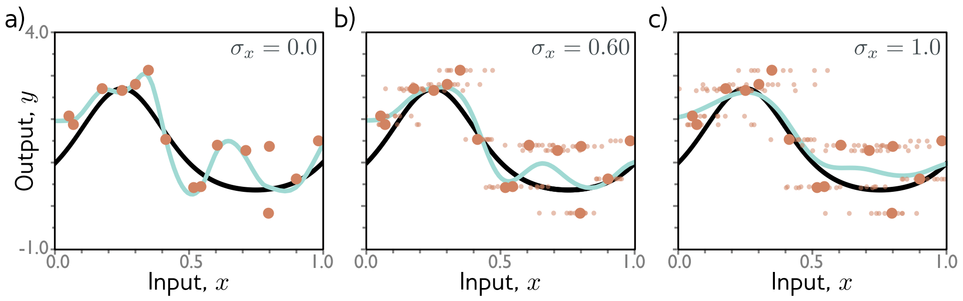


Figure 9.10 Adding noise to inputs. At each step of SGD random noise with variance σ_x^2 is added to the batch data. a-c) Fitted model with different levels of noise (small dots represent ten samples). Adding more noise smooths out the fitted function (cyan line).

shown to be equivalent to adding a regularizing term that penalizes the derivatives of the output of the network with respect to the input. An extreme version of this is *adversarial training*, in which the optimization algorithm actively searches for small perturbations of the input that cause large changes to the desired output. These can be thought of as worst-case additive noise vectors.

A second possibility is to add noise to the weights. This encourages the network to make sensible predictions even for small perturbations of the weights. The result is that the training converges to local minima that are in the middle of wide, flat regions, where changing the individual weights does not matter much.

Finally, we can perturb the labels. The maximum-likelihood criterion for multi-class classification aims to predict the correct class with absolute certainty (equation 5.27). To do this, the final network activations (*i.e.*, before the softmax function) must be pushed to very large values for the correct class and very small values for the wrong classes.

Problem 9.4

We could discourage this overconfident behavior by assuming that a proportion ρ of the training labels are incorrect and belong with equal probability to the other classes. In principle, this could be done by randomly changing the labels at each iteration of training. However, the same end can be achieved by changing the loss function so that it minimizes the cross entropy between the predicted distribution and the distribution where the ground truth label has probability $1 - \rho$ and the other classes have equal probability. This is known as *label smoothing* and improves generalization in a diverse variety of scenarios.

9.3.5 Bayesian approaches

The maximum likelihood approach is generally overconfident; in the training phase, it selects the most likely parameters and bases its predictions on the model defined

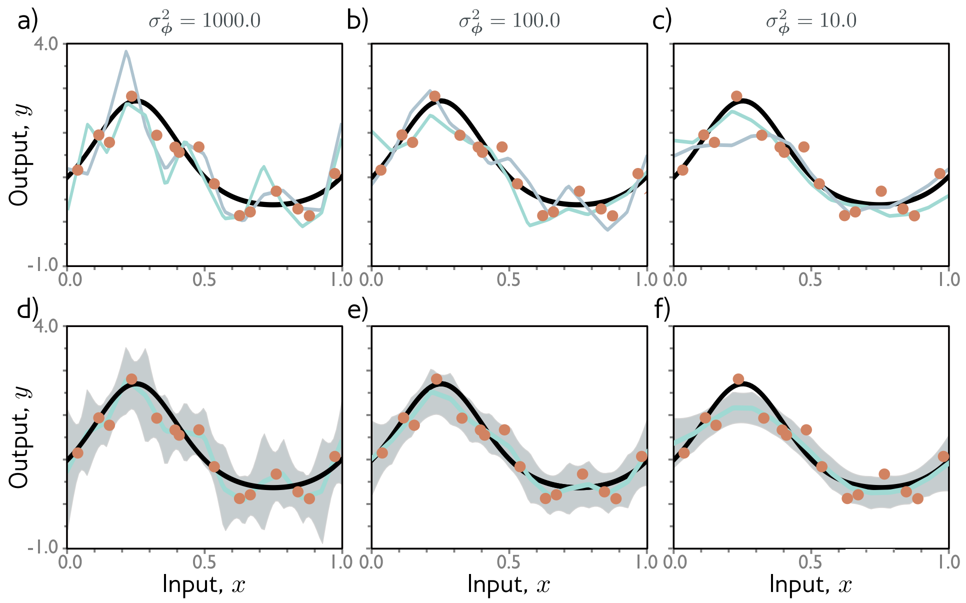


Figure 9.11 Bayesian approach applied to simplified network model (figure 8.4). Rather than finding a single set of parameters, the Bayesian approach treats the parameters as uncertain. The posterior probability $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ of a given set of parameters is determined by their compatibility with the data and a prior distribution $Pr(\boldsymbol{\phi})$. a-c) Two possible sets of parameters (cyan curves) sampled from the posterior distribution using normally distributed priors with mean zero and three different variances. When the prior variance is small, the parameters also tend to be small, and the sampled functions become smoother. d-f) Inference proceeds by taking a weighted sum over all possible parameter values where the weights are given by their probabilities. Accordingly, we get both a mean prediction (cyan curves) and an associated uncertainty (gray region represents two standard deviations).

by these. However, it may be that many parameter values are broadly compatible with the data and are only slightly less likely. The Bayesian approach treats the parameters as unknown variables and during training computes a distribution $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ over these parameters $\boldsymbol{\phi}$ conditioned on the training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using Bayes's rule:

$$Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) = \frac{\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi})Pr(\boldsymbol{\phi})}{\int \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi})Pr(\boldsymbol{\phi})d\boldsymbol{\phi}}, \tag{9.11}$$

where $Pr(\boldsymbol{\phi})$ is the prior probability of the parameters, and the denominator is a normalizing term. Accordingly, every possible set of parameters in the model family is assigned a probability (figure 9.11).

The final prediction $\mathbf{y}$ for new input $\mathbf{x}$ is an infinite weighted sum (*i.e.*, an integral) of the predictions for each parameter value, where the weights are the associated probabilities:

$$Pr(\mathbf{y}|\mathbf{x},\{\mathbf{x}_i,\mathbf{y}_i\}) = \int Pr(\mathbf{y}|\mathbf{x},\boldsymbol{\phi})Pr(\boldsymbol{\phi}|\{\mathbf{x}_i,\mathbf{y}_i\})d\boldsymbol{\phi}. \tag{9.12}$$

One way to think of this is that it is an infinite weighted ensemble of models, where the weight depends on (i) their agreement with the data and (ii) the prior probability of the parameters.

The Bayesian approach is elegant and can provide more robust predictions than those trained with maximum likelihood. Unfortunately, for complex models like neural networks, there is no practical way to represent the full probability distribution over the parameters or to integrate over it during the inference phase. Consequently, all current methods of this type make approximations of some kind, and typically these add considerable complexity to learning and inference.

9.3.6 Transfer learning and multi-task learning

When the training data are limited, it is possible to exploit other datasets to improve model performance. In *transfer learning* (figure 9.12a), the network is trained to perform a related secondary task where the data are more plentiful. The resulting model is subsequently adapted to the real problem. This is typically done by removing the last layer and adding one or more layers that produce a suitable output for the real problem. The first part of the model may be fixed, and the new layers trained, or we may *fine-tune* the entire model.

The core idea is that network will build a good internal representation of the data from the secondary task, and this can subsequently be exploited for the primary task. Equivalently, transfer learning can be viewed as initializing most of the parameters of the final network in a sensible part of the space that is likely to produce a good solution.

A related technique is *multi-task* learning (figure 9.12b), in which the network is trained to solve several problems concurrently. For example, the network might take an image and simultaneously learn to segment the scene, estimate the pixel-wise depth, and predict a caption describing the image. All these tasks require some common understanding of the image and so when learned simultaneously, the model performance may improve for each.

9.3.7 Self-supervised learning

The above discussion assumes that we have plentiful data for a secondary task or data for multiple tasks to be learned concurrently. If not, then we can create

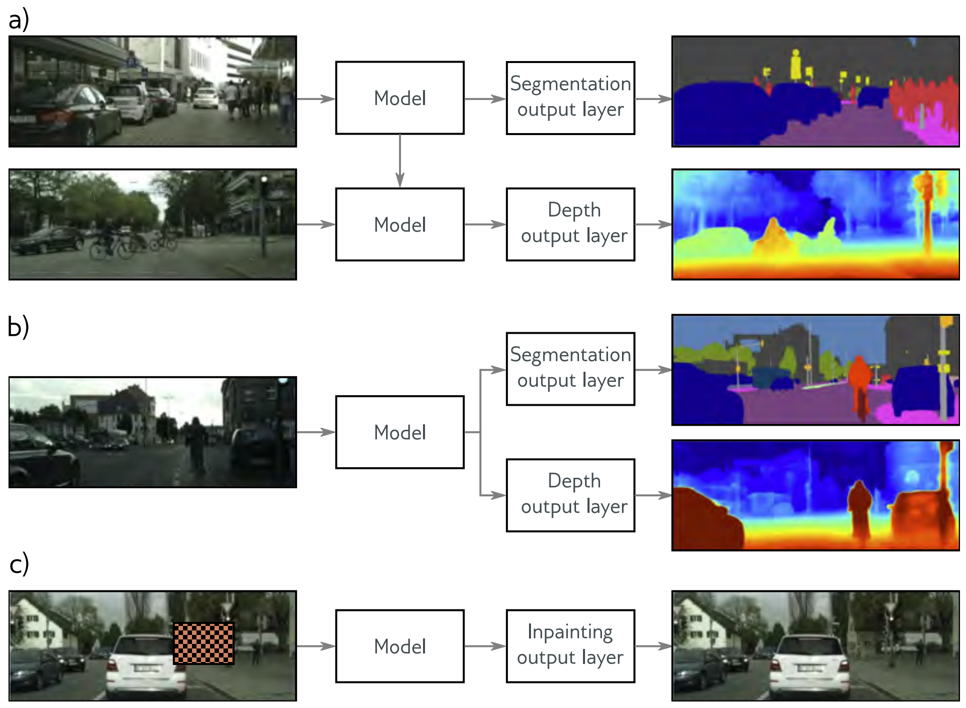


Figure 9.12 Transfer learning, multi-task learning, and self-supervised learning. a) Transfer learning is used when we have limited labeled data for the primary task (here depth estimation), but plentiful data for an associated secondary task (here segmentation). We train a model for the secondary task, remove the final layers of this model, and replace them with new layers appropriate to the primary task. We then train either just the new layers or continue to train the entire network. The network should learn a good internal representation of the data from the secondary task that can be exploited for the primary task. b) In multi-task learning, we simultaneously train a model to perform multiple tasks, with the idea that performance on each task will improve. c) In generative self-supervised learning, we remove part of the data and train the network to complete the missing information. This permits transfer learning when no labels are available. Images from Cordts *et al.* (2016).

large amounts of 'free' labeled data using *self-supervised* learning and use this for transfer learning. There are two families of methods for self-supervised learning:

Generative methods: Part of each data example is systematically masked, and the secondary task is then to predict the missing part (figure 9.12c). For example, we might use a corpus of unlabeled images and with a secondary task that aims to *inpaint* (fill in) missing parts of the image (figure 9.12c). Similarly, we might use a large corpus of text and then remove one or two words from each sentence. We train a network to predict the missing words and then fine-tune this network for the real language task that we are interested in (see chapter 12).

Contrastive methods: Two versions of each unlabeled example are presented, where one has been distorted in some way. The system is trained to predict which is the original version. For example, we might use a corpus of unlabeled images where the secondary task is to determine which version of the image is upside-down. Similarly, we might use a large corpus of text, where the secondary task is to determine whether two sentences followed each other in the original text or were taken from different places.

9.3.8 Augmentation

Transfer learning improves performance by exploiting a different dataset. Multi-task learning improves performance by exploiting additional labels. A third option is to expand the original dataset. For many tasks, we can transform each input data example in such a way that the label stays the same. For example, an image classification task might aim to determine if there is a bird in an image (figure 9.13). Here, we could rotate, flip, blur, or manipulate the color balance of the image and the label remains valid. Similarly, for tasks where the input is text, we can substitute synonyms or translate to another language and back again. For tasks where the input is audio, we can amplify or attenuate different frequency bands.

Generating extra training examples using this approach is known as *data augmentation*. The aim is to teach the machine learning system to be indifferent to these irrelevant transformations of the data.

9.4 Summary

Explicit regularization involves adding an extra term to the loss function that changes the position of the minimum. The term can be interpreted a prior probability over the parameters. Stochastic gradient descent with a finite step size does not neutrally descend to the minimum of the loss function. This bias can be interpreted as adding additional terms to the loss function, and this is known as implicit regularization.

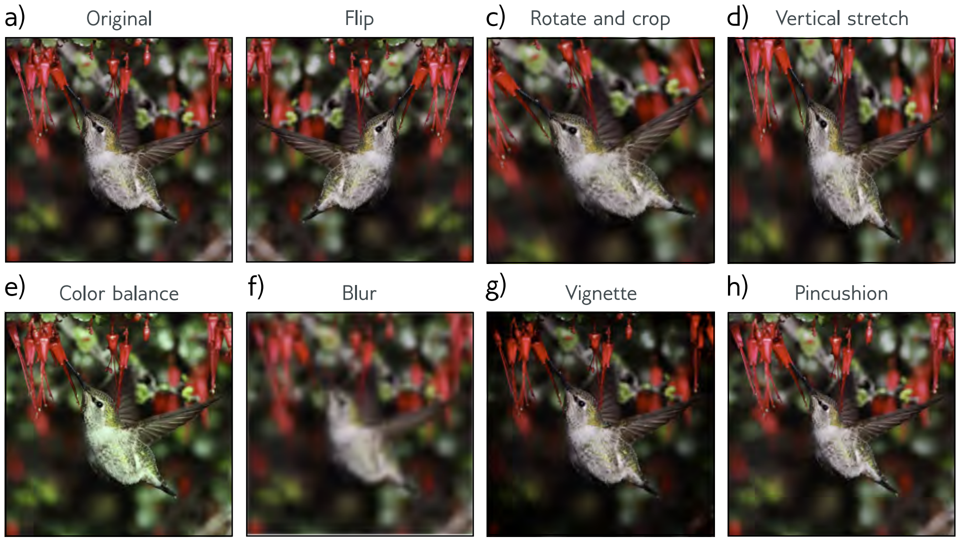


Figure 9.13 Data augmentation. For certain types of problem, it is easy to transform each data example multiple times to augment the dataset. a) Original image. b-h) Various geometric and photometric transformations of this image. For an image classification problem, all these modified versions still have the same label 'bird'. Adapted from Wu *et al.* (2015).

There are also many heuristics that are intended to improve the generalization performance of deep learning methods, including early stopping, dropout, ensembling, the Bayesian approach, adding noise, transfer learning, multi-task learning, and data augmentation. Considered together, there are four main principles behind these methods (figure 9.14). We can (i) encourage the function to be smoother (*e.g.*, L2 regularization), (ii) increase the effective amount of data (*e.g.*, data augmentation), (iii) combine multiple models (*e.g.*, ensembling), or (iv) search for wider minima in the loss function (*e.g.*, applying noise to network weights).

One further way to improve model performance is to share parameters between different parts of the model. For example, in an image segmentation task, it is inefficient to separately learn which pattern of pixels constitutes a tree at every location in the image. This is the topic of chapter 10.

Notes

An overview and taxonomy of regularization techniques in deep learning can be found in Kukačka *et al.* (2017). Notably missing from the discussion in this chapter is BatchNorm (Szegedy *et al.* 2016), which is described in chapter 11.

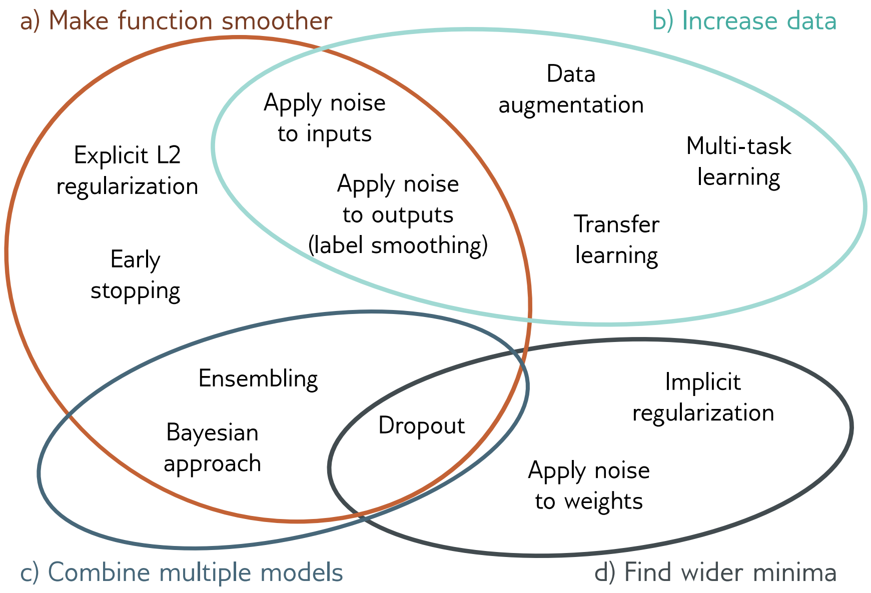


Figure 9.14 Regularization methods. The regularization methods discussed in this chapter aim to improve generalization by one of four mechanisms. a) Some methods aim to make the modeled function smoother. b) Other methods increase the effective amount of data. c) The third group of methods combine multiple models and hence mitigate against uncertainty in the fitting process. d) Finally, the fourth group of methods encourages the training process to converge to a wide minimum where small errors in the estimated parameters are less important.

Regularization: L2 regularization penalizes the sum of squares of the network weights. This encourages the output function to change slowly (*i.e.*, become smoother) and is the most used regularization term. It is sometimes referred to as Frobenius norm regularization as it penalizes the Frobenius norm of the weight matrices. It is often also mistakenly referred to as "weight decay" although this is a separate technique devised by Hanson & Pratt (1988) in which the parameters $\boldsymbol{\phi}$ are updated as:

$$\boldsymbol{\phi} \longleftarrow (1-\lambda')\boldsymbol{\phi} - \alpha\frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.13}$$

where as usual α is the learning rate and L is the loss. This is identical to gradient descent, except that the weights are reduced by a factor of $1-\lambda'$ before the gradient update. For standard SGD, weight decay is equivalent to L2 regularization (equation 9.5) with coefficient $\lambda = \lambda'/2\alpha$. However, for Adam, the learning rate α is different for each parameter and so L2 regularization and weight decay differ. Loshchilov & Hutter (2017) present a modified version of Adam called AdamW that implements weight decay correctly and show that this improves performance.

Problem 9.5

A different approach is to encourage sparsity in the weights. The L0 regularization term

Appendix C.1.5 vector norms

applies a fixed penalty for every non-zero weight. The overall effect is to 'prune' the network. Another approach is to add an L0 regularization term that encourages group sparsity; this might apply a fixed penalty if any of the weights contributing to a given neuron are non-zero. If they are all zero, we can remove this neuron, decreasing the model size, and making inference faster.

Unfortunately, L0 regularization is challenging to implement since the derivative of the regularization term is not smooth, and more sophisticated fitting methods are required (see Louizos *et al.* 2018). Somewhere between L2 and L0 regularization is L1 regularization or *LASSO* (least absolute shrinkage and selection operator), which imposes a penalty on the absolute values of the weights. L2 regularization somewhat discourages sparsity in that the derivative of the squared penalty decreases as the weight becomes smaller, lowering the pressure to make it smaller still. L1 regularization does not have this disadvantage as the derivative of the penalty is constant. This can produce sparser solutions than L2 regularization but is much easier to optimize than L0 regularization. Sometimes both L1 and L2 regularization terms are both used, and this is termed an *elastic net* penalty.

Problem 9.6

An alternative approach to regularization is to directly modify the gradients of the learning algorithm without ever explicitly formulating a modified loss function (as we already saw in equation 9.13). This approach has been used to promote sparsity during backpropagation (Schwarz *et al.* 2021).

The evidence on the effectiveness of explicit regularization is mixed. Zhang *et al.* (2016a) showed that L2 regularization contributes little to generalization. It has been shown that the Lipschitz constant of the network (how fast the function can change as we modify the input) bounds the generalization error (Bartlett *et al.* 2017a; Neyshabur *et al.* 2017b). However, the Lipschitz constant depends on the product of the spectral norm of the weight matrices $\mathbf{\Omega}_k$, which are only indirectly dependent on the magnitudes of the individual weights. Bartlett *et al.* (2017a), Neyshabur *et al.* (2017b), and Yoshida & Miyato (2017) all add terms that indirectly encourage the spectral norms to be smaller. Gouk *et al.* (2018) take a different approach and develop an algorithm that constrains the Lipschitz constant of the network to be below a certain value.

Appendix C.4
Lipschitz constant

Appendix C.1.6
spectral norm

Implicit regularization in gradient descent: The gradient descent step is

$$\boldsymbol{\phi}_1 = \boldsymbol{\phi}_0 + \alpha \mathbf{g}[\boldsymbol{\phi}_0], \tag{9.14}$$

where $\mathbf{g}[\boldsymbol{\phi}_0]$ is the negative of the gradient of the loss function and α is the step size. As $\alpha \to 0$, the gradient descent process can be described by a differential equation:

$$\frac{\partial \boldsymbol{\phi}}{\partial t} = \mathbf{g}[\boldsymbol{\phi}]. \tag{9.15}$$

For typical step sizes α, the discrete and continuous versions converge to different solutions. We can apply a technique called *backward error analysis* to find a correction $\mathbf{g}_1[\boldsymbol{\phi}]$ to the continuous version:

$$\frac{\partial \boldsymbol{\phi}}{\partial t} \approx \mathbf{g}[\boldsymbol{\phi}] + \alpha \mathbf{g}_1[\boldsymbol{\phi}] + \ldots, \tag{9.16}$$

so that it gives the same result as the discrete version.

Consider the first two terms of a Taylor expansion of the modified continuous solution $\boldsymbol{\phi}$ around initial position $\boldsymbol{\phi}_0$:

$$\begin{aligned}
\boldsymbol{\phi}[\alpha] &\approx \boldsymbol{\phi} + \alpha\frac{\partial\boldsymbol{\phi}}{\partial t} + \frac{\alpha^2}{2}\frac{\partial^2\boldsymbol{\phi}}{\partial t^2}\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha\left(\mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2}\left(\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\frac{\partial\boldsymbol{\phi}}{\partial t} + \alpha\frac{\partial\mathbf{g}_1[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\frac{\partial\boldsymbol{\phi}}{\partial t}\right)\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&= \boldsymbol{\phi} + \alpha\left(\mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2}\left(\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}] + \alpha\frac{\partial\mathbf{g}_1[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]\right)\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha\mathbf{g}[\boldsymbol{\phi}] + \alpha^2\left(\mathbf{g}_1[\boldsymbol{\phi}] + \frac{1}{2}\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]\right)\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0}, \qquad (9.17)
\end{aligned}$$

where in the second line, we have introduced the correction term (equation 9.16), and in the final line, we have removed terms of greater order than α^2.

Note that the first two terms on the right-hand side $\boldsymbol{\phi}_0 + \alpha\mathbf{g}[\boldsymbol{\phi}_0]$ are the same as the discrete update (equation 9.14). Hence, to make the continuous and discrete versions arrive at the same place, the second term on the right-hand side must equal zero allowing us to solve for $\mathbf{g}_1[\boldsymbol{\phi}]$:

$$\mathbf{g}_1[\boldsymbol{\phi}] = -\frac{1}{2}\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]. \qquad (9.18)$$

When we optimize neural networks, the evolution function $\mathbf{g}[\boldsymbol{\phi}]$ is the negative of the gradient of the loss so we have:

$$\begin{aligned}
\frac{\partial\boldsymbol{\phi}}{\partial t} &\approx \mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}] \\
&= -\frac{\partial L}{\partial\boldsymbol{\phi}} - \frac{\alpha}{2}\left(\frac{\partial^2 L}{\partial\boldsymbol{\phi}^2}\right)\frac{\partial L}{\partial\boldsymbol{\phi}}. \qquad (9.19)
\end{aligned}$$

This is the equivalent to performing continuous gradient descent on the loss function:

$$\mathrm{L}_{GD}[\boldsymbol{\phi}] = \mathrm{L}[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial\boldsymbol{\phi}}\right\|^2, \qquad (9.20)$$

because we retrieve equation 9.19 by taking the derivative of equation 9.20.

This formulation of implicit regularization was developed by Barrett & Dherin (2021) and extended to stochastic gradient descent by Smith *et al.* (2021). Smith *et al.* (2020) and others have shown that stochastic gradient descent with small or moderate batch sizes outperforms full batch gradient descent on the test set, and this may in part be due to implicit regularization.

Early stopping: Bishop (1995) and Sjöberg & Ljung (1995) argued that the effect of early stopping was to limit the effective space of solutions that the training procedure can explore; given that the weights are initialized to small values, this leads to the idea that early stopping helps prevent the weights from getting too large. Goodfellow *et al.* (2016) show that under a quadratic approximation of the loss function with parameters initialized to zero, early stopping is equivalent to L2 regularization in gradient descent. The effective regularization weight λ can be shown to be approximately $1/(\tau\alpha)$ where α is the learning rate and τ is the early stopping time.

Ensembling: Ensembles can be trained by using different random seeds (Lakshminarayanan *et al.* 2017), hyperparameters (Wenzel *et al.* 2020b), or even entirely different families of model. The models can be combined by averaging their predictions, weighting the predictions, or *stacking* (Wolpert 1992), in which the results are combined using another machine learning model. Lakshminarayanan *et al.* (2017) showed that averaging the output of independently trained networks can improve accuracy, calibration, and robustness. However, Frankle *et al.* (2020) showed that if we average together the weights to make one model, then the network fails. Fort *et al.* (2020) compared ensembling solutions that resulted from different initializations with ensembling solutions that were generated from the same original mode. For example, in the latter case, they consider exploring around the solution in a limited subspace to find other good nearby points. They found that both techniques provide complementary benefits, but that genuine ensembling from different random starting points provides a bigger benefit.

Appendix C.1.3 subspaces

An efficient way of ensembling is to combine models from the intermediate stages of training. To this end, Izmailov *et al.* (2018) introduce *stochastic weight averaging*, in which the model weights are sampled at different time steps and averaged together. As the name suggests, *snapshot ensembles* (Huang *et al.* 2017) also store the models from different time steps and average their predictions. To increase the diversity of these models, they cyclically increase and decrease the learning rate. Garipov *et al.* (2018) observed that different minima of the loss function are often connected by a low-energy path. Motivated by this observation they developed a method that explores low-energy regions around an initial solution to provide diverse models without having to completely retrain. This method is known as *fast geometric ensembling.* A review of ensembling methods for deep learning can be found in Ganaie *et al.* (2021).

Dropout: Dropout was first introduced by Hinton *et al.* (2012b) and Srivastava *et al.* (2014). Dropout is applied at the level of hidden units. Dropping a hidden unit has the same effect as temporarily setting all the incoming weights and the bias to zero. Wan *et al.* (2013) generalized dropout by randomly setting weights to zero.

Gal & Ghahramani (2015) and Kendall & Gal (2017) proposed Monte-Carlo dropout, in which inference is computed with several dropout patterns and the results are averaged together. Gal & Ghahramani (2015) argued that this can be interpreted as Bayesian inference.

Dropout is equivalent to applying multiplicative Bernoulli noise to the hidden units. Similar benefits have been shown to be derived from using other noise distributions including the normal distribution (Srivastava *et al.* 2014; Shen *et al.* 2017), uniform distribution (Shen *et al.* 2017), and beta distribution (Liu *et al.* 2019).

Adding noise: Bishop (1995) and An (1996) both added Gaussian noise to the network inputs to improve performance. Bishop (1995) showed that this is equivalent to weight decay. An (1996) also investigated adding noise to the weights. DeVries & Taylor (2017) added Gaussian noise to the hidden units. Xu *et al.* (2015) apply noise in a different way with the *randomized ReLU* by making the activation functions stochastic.

Finding wider minima: It is thought that wider minima in the loss function generalize better. Here, the exact values of the weights are less important and so performance should be robust to errors in their estimates. One of the reasons that applying noise to parts of the network during training is effective is that it encourages the network to be indifferent to their exact values.

Chaudhari *et al.* (2017) develop a variant of SGD that deliberately biases the optimization towards flat minima, which they call *entropy SGD.* The basic idea is to incorporate the local entropy as a term in the loss function. In practice, this takes the form of one SGD

Appendix B.1.3 entropy

like update within another. Keskar *et al.* (2017) showed that SGD finds wider minima as the batch size is reduced. This may be because of the batch variance term that results from implicit regularization by SGD.

Ishida *et al.* (2020) use a technique named *flooding*, in which they intentionally prevent the training loss from becoming zero. This encourages the solution to perform a random walk over the loss landscape and drift into a flatter area with better generalization.

Label smoothing: Label smoothing was introduced by Szegedy *et al.* (2016) in the context of image classification, but has since been shown to provide performance improvements in other areas including speech recognition (Chorowski & Jaitly 2016), machine translation (Vaswani *et al.* 2017) and language modeling (Pereyra *et al.* 2017). The precise mechanism by which label smoothing improves test performance is not well understood although Müller *et al.* (2019) show that it improves the calibration of the predicted output probabilities. A closely related technique is *DisturbLabel* (Xie *et al.* 2016), in which a certain percentage of the labels in each batch are randomly switched at each iteration of training.

Bayesian approaches: For some models, including the simplified neural network model in figure 9.11, the Bayesian predictive distribution can be computed in closed form. This is explained in detail in Bishop (2006) and Prince (2012). For neural networks, the posterior distribution over the parameters cannot be represented in closed form and must be approximated. The two main approaches are variational Bayes (Hinton & van Camp 1993; MacKay *et al.* 1995; Barber & Bishop 1997; Blundell *et al.* 2015), in which the posterior is approximated by a simpler tractable distribution, and Markov Chain Monte Carlo (MCMC) methods, which approximate the distribution by drawing a set of samples (Neal 1995; Welling & Teh 2011; Chen *et al.* 2014; Ma *et al.* 2015; Li *et al.* 2016a). The generation of samples can be integrated into the stochastic gradient descent algorithm, and this is known as stochastic gradient MCMC (see Ma *et al.* 2015). It has recently been discovered that 'cooling' the posterior distribution over the parameters (making it sharper) improves the predictions from these models (Wenzel *et al.* 2020a), but this is not currently fully understood (see Noci *et al.* 2021).

Transfer learning: Transfer learning for visual tasks works extremely well (Sharif Razavian *et al.* 2014) and has supported rapid progress in computer vision including the original AlexNet results (Krizhevsky *et al.* 2012). Transfer learning has also had an enormous impact on natural language processing where many models are based on pre-trained features from the BERT language model (Devlin *et al.* 2018). More information about transfer learning can be found in Zhuang *et al.* (2020).

Self-supervised learning: Self-supervised learning techniques for images have included inpainting masked image regions (Pathak *et al.* 2016), predicting the relative position of patches in an image (Doersch *et al.* 2015), re-arranging permuted image tiles back into their original configuration (Noroozi & Favaro 2016), colorizing black and white image (Zhang *et al.* 2016c), and transforming rotated images back to their original orientation (Gidaris *et al.* 2018). In SimCLR (Chen *et al.* 2020), a network is learned that maps versions of the same image that have been photometrically and geometrically transformed to the same representation, while repelling versions of different images, with the goal of becoming indifferent to irrelevant image transformations. A survey of self-supervised learning in images can be found in Jing & Tian (2020).

Self-supervised learning in natural language processing might be based on masking words from a sentence (Devlin *et al.* 2018), predicting the next word in a sentence (Radford *et al.* 2019; Brown *et al.* 2020), and predicting whether two sentences follow one another (Devlin *et al.* 2018). In the field of automatic speech recognition, the Wav2Vec model (Schneider *et al.* 2019) aims to distinguish an original audio sample from one where 10ms of audio has been swapped out from elsewhere in the clip. Self-supervision has also been applied to graph neural networks (chapter 13) and tasks include recovering masked features (You *et al.* 2020) and recovering the adjacency structure of the graph (Kipf & Welling 2016). A review of graph self-supervised learning can be found in Liu *et al.* (2021b).

Data augmentation: Data augmentation for images dates back to at least LeCun *et al.* (1998a), and contributed to the success of AlexNet (Krizhevsky *et al.* 2012), in which the dataset was increased by a factor of 2048. Typical augmentation approaches for images include geometric transformations, changing or manipulating the color space, noise injection, and applying spatial filters. More elaborate techniques include randomly mixing images (Inoue 2018; Summers & Dinneen 2019), randomly erasing parts of the image (Zhong *et al.* 2020), style transfer (Jackson *et al.* 2019), and randomly swapping image patches (Kang *et al.* 2017). In addition, many studies have used generative adversarial networks or GANs (see chapter 16) to produce novel, but plausible data examples (*e.g.*, Calimeri *et al.* 2017). In other cases, the data have been augmented with adversarial examples (Goodfellow *et al.* 2014), which are minor perturbations of the training data that cause the example to be misclassified. A comprehensive review of data augmentation for images can be found in Shorten & Khoshgoftaar (2019).

Augmentation methods for acoustic data include pitch shifting, time stretching, dynamic range compression, and adding random noise (*e.g.*, (Abeßer *et al.* 2017; Salamon & Bello 2017; Xu *et al.* 2015; Lasseck 2018), as well as mixing data pairs (Zhang *et al.* 2017; Yun *et al.* 2019), masking features (Park *et al.* 2019), and using GANs to generate new data (Mun *et al.* 2017). Augmentation for speech data includes vocal tract length perturbation (Jaitly & Hinton 2013; Kanda *et al.* 2013), style transfer (Gales 1998; Ye & Young 2004), adding noise (Hannun *et al.* 2014), and synthesizing speech (Gales *et al.* 2009).

Augmentation methods for text data include adding noise at a character level by switching, deleting, and inserting letters (Belinkov & Bisk 2017; Feng *et al.* 2020), or by generating adversarial examples (Ebrahimi *et al.* 2017), using common spelling mistakes (Coulombe 2018), randomly swapping or deleting words (Wei & Zou 2019), using synonyms (Kolomiyets *et al.* 2011), altering adjectives (Li *et al.* 2017), passivization (Min *et al.* 2020), using generative models to create new data (Qiu *et al.* 2020), and round-trip translation to another language and back again (Aiken & Park 2010). A review of augmentation methods for text data can be found in Bayer *et al.* (2021).

Problems

Problem 9.1 Consider a model where the prior distribution over the parameters is a normal distribution with mean zero and variance $\sigma^2_{\boldsymbol{\phi}}$ so that

$$Pr(\boldsymbol{\phi}) = \prod_{k=1}^{K} \text{Norm}_{\phi_k}[0, \sigma^2_{\boldsymbol{\phi}}]. \tag{9.21}$$

where k indexes the parameters of the model. When we apply a prior, we maximize $\prod_{i=1}^{I} Pr(\mathbf{x}_i|\boldsymbol{\phi})Pr(\boldsymbol{\phi})$. Show that the associated loss function of this model is equivalent to L2 regularization.

Problem 9.2 How do the gradients in the backpropagation algorithm (section 7.2.2) change when L2 regularization (equation 9.5) is added?

Problem 9.3 Consider a univariate linear regression problem $y = \phi_0 + \phi_1 x$ where x is the input, y is the output, and ϕ_0 and ϕ_1 are the intercept and slope parameters respectively. Assume that we have I data examples $\{x_i, y_i\}$ and are using a least squares loss function. Consider adding Gaussian noise with mean 0 and variance σ_x^2 to the inputs x_i during each iteration of training. What is the expected gradient update?

Problem 9.4 Derive the loss function for multi-class classification when we use label smoothing so that the target probability distribution has 0.9 at the correct class and the remaining probability mass of 0.1 is divided between the remaining $D_o - 1$ classes.

Problem 9.5 Show that the weight decay parameter update with decay rate λ':

$$\boldsymbol{\phi} \longleftarrow (1-\lambda')\boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.22}$$

on the original loss function $\mathrm{L}[\boldsymbol{\phi}]$ is equivalent to a standard gradient update using L2 regularization so that the modified loss function $\tilde{\mathrm{L}}[\boldsymbol{\phi}]$ is:

$$\tilde{\mathrm{L}}[\boldsymbol{\phi}] = \mathrm{L}[\boldsymbol{\phi}] + \frac{\lambda'}{2\alpha}\sum_k \phi_k^2, \tag{9.23}$$

where $\boldsymbol{\phi}$ are the parameters, and α is the learning rate.

Problem 9.6 Consider a model with two parameters $\boldsymbol{\phi} = [\boldsymbol{\phi}_0, \boldsymbol{\phi}_1]^T$. Draw the $L0$, $L\frac{1}{2}$, $L1$, and $L2$ regularization terms in a similar form to that shown in figure 9.1b.

Chapter 10

Convolutional networks

Chapter 11

Residual networks and BatchNorm

Chapter 12

Transformers

Chapter 13

Graph neural networks

Chapter 14

Variational auto-encoders

Chapter 15

Normalizing flows

Good description of reversible components plus applications to CNNs in (Mangalam *et al.* 2022) Reversible Image Transformers.

Don't need to cache activations in reversible networks. – memory efficient.

Chapter 16

Generative adversarial networks

Chapter 17

Diffusion models

Chapter 18

Deep reinforcement learning

Chapter 19

Why does deep learning work?

Appendix A

Notation

This is a brief guide to the notational conventions used in this text.

Appendix B

Probability

Appendix C

Maths

C.1 Linear algebra

C.1.1 Trace

C.1.2 Determinant

C.1.3 Subspace

C.1.4 Eigenvalues

C.1.5 Vector norms

C.1.6 Spectral norm

C.1.7 Permutation matrices

C.2 Matrix calculus

C.3 Autocorrelation functions

C.4 Lipschitz constant

Bibliography

Abesser, J., Mimilakis, S. I., Gräfe, R., Lukashevich, H., & Fraunhofer, I. (2017) Acoustic scene classification by combining autoencoder-based dimensionality reduction and convolutional neural networks. In *Proc. of the Detection and Classification of Acoustic Scenes and Events 2017 Workshop (DCASE2017)*, pp. 7–11. 156

Aggarwal, C. C., Hinneburg, A., & Keim, D. A. (2001) On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pp. 420–434. Springer. 128

Aiken, M., & Park, M. (2010) The efficacy of round-trip translation for mt evaluation. *Translation Journal* **14** (1): 1–10. 156

An, G. (1996) The effects of adding noise during backpropagation training on a generalization performance. *Neural computation* **8** (3): 643–674. 154

Arora, R., Basu, A., Mianjy, P., & Mukherjee, A. (2016) Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491* . 45

Barber, D., & Bishop, C. (1997) Ensemble learning for multi-layer networks. *Advances in neural information processing systems* **10**. 155

Barrett, D. G. T., & Dherin, B., (2021) Implicit gradient regularization. 153

Barron, J. T., (2019) A general and adaptive robust loss function. 66

Bartlett, P. L., Foster, D. J., & Telgarsky, M. J. (2017a) Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pp. 6240–6249. 152

Bartlett, P. L., Harvey, N., Liaw, C., & Mehrabian, A., (2017b) Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. 127

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M., (2018) Automatic differentiation in machine learning: a survey. 105

Bayer, M., Kaufhold, M.-A., & Reuter, C. (2021) A survey on data augmentation for text classification. *arXiv preprint arXiv:2107.03158* . 156

Belinkov, Y., & Bisk, Y. (2017) Synthetic and natural noise both break neural machine translation. *arXiv preprint arXiv:1711.02173* . 156

Belkin, M., Hsu, D., Ma, S., & Mandal, S. (2019) Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences* **116** (32): 15849–15854. 123, 127

Bergstra, J., & Bengio, Y. (2012) Random search for hyper-parameter optimization. *Journal of machine learning research* **13** (2). 129

Bergstra, J. S., Bardenet, R., Bengio, Y., & Kégl, B. (2011) Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554. 129

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999) When is "nearest neighbor" meaningful? In *International conference on database theory*, pp. 217–235. Springer. 128

Bishop, C. M. (1994) Mixture density networks. 67

Bishop, C. M. (1995) Regularization and complexity control in feed-forward networks. 153, 154

Bishop, C. M. (2006) *Pattern recognition and machine learning*. Springer. 155

Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015) Weight uncertainty in neural network. In *International conference on machine learning*, pp. 1613–1622. PMLR. 155

Bottou, L. (2012) *Stochastic Gradient Descent Tricks*, pp. 421–436. Springer Berlin Heidelberg. 86

Bottou, L., Curtis, F. E., & Nocedal, J., (2018) Optimization methods for large-scale machine learning. 86

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., & others. (2020) Language models are few-shot learners. *Advances in neural information processing systems* **33**: 1877–1901. 155

Bryson, A., Ho, Y.-C., & Siouris, G. (1979) Applied optimal control: Optimization, estimation, and control. *Systems, Man and Cybernetics, IEEE Transactions on* **9**: 366 – 367. 105

Bubeck, S., & Sellke, M., (2021) A universal law of robustness via isoperimetry. 128

Burda, Y., Grosse, R., & Salakhutdinov, R., (2016) Importance weighted autoencoders. 66

Buschjäger, S., & Morik, K., (2021) There is no double-descent in random forests. 127

Calimeri, F., Marzullo, A., Stamile, C., & Terracina, G. (2017) Biomedical data augmentation using adversarial neural networks. In *International conference on artificial neural networks*, pp. 626–634. Springer. 156

Cao, Z., Qin, T., Liu, T.-Y., Tsai, M.-F., & Li, H. (2007) Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pp. 129–136. 67

Cauchy, A. (1847) Methode generale pour la resolution des systemes d'equations simultanees. *C.R. Acad. Sci. Paris* **25**: 536–538. 86

Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J., Sagun, L., & Zecchina, R., (2017) Entropy-sgd: Biasing gradient descent into wide valleys. 154

Chen, T., Fox, E., & Guestrin, C. (2014) Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pp. 1683–1691. PMLR. 155

Chen, T., Kornblith, S., Norouzi, M., & Hinton, G. (2020) A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR. 155

Chen, T., Xu, B., Zhang, C., & Guestrin, C., (2016) Training deep nets with sublinear memory cost. 106

Chen, W., Liu, T.-Y., Lan, Y., Ma, Z.-M., & Li, H. (2009) Ranking measures and loss functions in learning to rank. *Advances in Neural Information Processing Systems* **22**: 315–323. 67

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2019) On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446* . 88

Chorowski, J., & Jaitly, N. (2016) Towards better decoding and language model integration in sequence to sequence models. *arXiv preprint arXiv:1612.02695* . 155

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015) Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289* . 28

Cohen, N., Sharir, O., & Shashua, A. (2015) On the expressive power of deep learning: A tensor analysis. *CoRR* **abs/1509.05009**. 46

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., & Schiele, B. (2016) The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3213–3223. 148

Coulombe, C. (2018) Text data augmentation made simple by leveraging nlp cloud apis. *arXiv preprint arXiv:1812.04718* . 156

Cristianini, M., & Shawe-Taylor, J. (2000) *An introduction to support vector machines.* Cambridge University Press. 67

Cybenko, G. (1989) Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* **2** (4): 303–314. 28

Dar, Y., Muthukumar, V., & Baraniuk, R. G., (2021) A farewell to the bias-variance tradeoff? an overview of the theory of overparameterized machine learning. 128

Dechter, R. (1986) Learning while searching in constraint-satisfaction-problems. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, p. 178–183. AAAI Press. 45

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* . 155

DeVries, T., & Taylor, G. W. (2017) Dataset augmentation in feature space. *arXiv preprint arXiv:1702.05538* . 154

Doersch, C., Gupta, A., & Efros, A. A. (2015) Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE international conference on computer vision*, pp. 1422–1430. 155

Domingos, P. (2000) A unified bias-variance decomposition. In *Proceedings of 17th international conference on machine learning*, pp. 231–238. Morgan Kaufmann Stanford. 127

Domke, J., (2010) Statistical machine learning. https://people.cs.umass.edu/ domke/. 109

Dorta, G., Vicente, S., Agapito, L., Campbell, N. D. F., & Simpson, I., (2018) Structured uncertainty prediction networks. 66

Dozat, T. (2016) Incorporating nesterov momentum into adam. 88

Duchi, J., Hazan, E., & Singer, Y. (2011) Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* **12** (Jul): 2121–2159. 87

Ebrahimi, J., Rao, A., Lowd, D., & Dou, D. (2017) Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751* . 156

Eldan, R., & Shamir, O. (2015) The power of depth for feedforward neural networks. *CoRR* **abs/1512.03965**. 46

Falkner, S., Klein, A., & Hutter, F. (2018) BOHB: robust and efficient hyperparameter optimization at scale. *CoRR* **abs/1807.01774**. 129

Fallah, N., Gu, H., Mohammad, K., Seyyedsalehi, S. A., Nourijelyani, K., & Eshraghian, M. R. (2009) Nonlinear poisson regression using neural networks: a simulation study. *Neural Computing and Applications* **18** (8): 939–943. 67

Fan, K., Li, B., Wang, J., Zhang, S., Chen, B., Ge, N., & Yan, Z. (2020) Neural zero-inflated quality estimation model for automatic speech recognition system. In *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, ed. by H. Meng, B. Xu, & T. F. Zheng, pp. 606–610. ISCA. 67

Feng, S. Y., Gangal, V., Kang, D., Mitamura, T., & Hovy, E. (2020) Genaug: Data augmentation for finetuning text generators. *arXiv preprint arXiv:2010.01794* . 156

Fort, S., Hu, H., & Lakshminarayanan, B., (2020) Deep ensembles: A loss landscape perspective. 154

Frankle, J., Dziugaite, G. K., Roy, D. M., & Carbin, M., (2020) Linear mode connectivity and the lottery ticket hypothesis. 154

Freund, Y., & Schapire, R. E. (1995) A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory: Eurocolt '95*, pp. 23–37. 67

Friedman, J. H. (1997) On bias, variance, 0/1—loss, and the curse-of-dimensionality. *Data mining and knowledge discovery* **1** (1): 55–77. 127

Fukushima, K. (1969) Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics* **5** (4): 322–333. 27

Gal, Y., & Ghahramani, Z., (2015) Dropout as a bayesian approximation: Representing model uncertainty in deep learning. 154

Gales, M. J. (1998) Maximum likelihood linear transformations for hmm-based speech recognition. *Computer speech & language* **12** (2): 75–98. 156

Gales, M. J., Ragni, A., AlDamarki, H., & Gautier, C. (2009) Support vector machines for noise robust asr. In *2009 IEEE Workshop on Automatic Speech Recognition & Understanding*, pp. 205–210. IEEE. 156

Ganaie, M., Hu, M., & others. (2021) Ensemble deep learning: A review. *arXiv preprint arXiv:2104.02395* . 154

Garipov, T., Izmailov, P., Podoprikhin, D., Vetrov, D. P., & Wilson, A. G. (2018) Loss surfaces, mode connectivity, and fast ensembling of dnns. *Advances in neural information processing systems* **31**. 154

Gidaris, S., Singh, P., & Komodakis, N. (2018) Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728* . 155

Glorot, X., & Bengio, Y. (2010) Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ed. by Y. W. Teh & M. Titterington, volume 9 of *Proceedings of Machine Learning Research*, pp. 249–256. PMLR. 105

Glorot, X., Bordes, A., & Bengio, Y. (2011) Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323. JMLR Workshop and Conference Proceedings. 27, 28

Goh, G. (2017) Why momentum really works. *Distill* . 87

Gomez, A. N., Ren, M., Urtasun, R., & Grosse, R. B. (2017) The reversible residual network: Backpropagation without storing activations. *Advances in neural information processing systems* **30**. 106

Goodfellow, I., Bengio, Y., & Courville, A. (2016) *Deep learning*. MIT press. 153

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014) Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* . 156

Gouk, H., Frank, E., Pfahringer, B., & Cree, M. (2018) Regularisation of neural networks by enforcing lipschitz continuity. *arXiv preprint arXiv:1804.04368* . 152

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., & He, K., (2018) Accurate, large minibatch sgd: Training imagenet in 1 hour. 88

Greydanus, S., (2020) Scaling down deep learning. 112

Griewank, A., & Walther, A. (2008) *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM. 105

Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., & others. (2014) Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* . 156

Hanson, S., & Pratt, L. (1988) Comparing biases for minimal network construction with back-propagation. *Advances in neural information processing systems* **1**. 151

He, K., Zhang, X., Ren, S., & Sun, J. (2015) Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR* **abs/1502.01852**. 28, 105

Hendrycks, D., & Gimpel, K. (2016) Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* . 28

Hinton, G., Srivastava, N., & Swersky, K., (2012a) Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. 87

Hinton, G., & van Camp, D. (1993) Keeping neural networks simple by minimising the description length of weights. 1993. In *Proceedings of COLT-93*, pp. 5–13. 155

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012b) Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* . 154

Hornik, K. (1991) Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4** (2): 251–257. 28

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., & others. (2019) Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1314–1324. 28

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017) Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109* . 154

Huang, X. S., Perez, F., Ba, J., & Volkovs, M. (2020) Improving transformer optimization through better initialization. In *International Conference on Machine Learning*, pp. 4475–4483. PMLR. 106

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., & Chen, Z., (2019) Gpipe: Efficient training of giant neural networks using pipeline parallelism. 106

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011) Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pp. 507–523. Springer. 129

Inoue, H. (2018) Data augmentation by pairing samples for images classification. *arXiv preprint arXiv:1801.02929* . 156

Ioffe, S., & Szegedy, C., (2015) Batch normalization: Accelerating deep network training by reducing internal covariate shift. 106

Ishida, T., Yamane, I., Sakai, T., Niu, G., & Sugiyama, M. (2020) Do we need zero training loss after achieving zero training error? *arXiv preprint arXiv:2002.08709* . 127, 155

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018) Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407* . 154

Jackson, P. T., Abarghouei, A. A., Bonner, S., Breckon, T. P., & Obara, B. (2019) Style augmentation: data augmentation via style randomization. In *CVPR Workshops*, volume 6, pp. 10–11. 156

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991) Adaptive mixtures of local experts. *Neural computation* **3** (1): 79–87. 67

Jaitly, N., & Hinton, G. E. (2013) Vocal tract length perturbation (vtlp) improves speech recognition. In *Proc. ICML Workshop on Deep Learning for Audio, Speech and Language*, volume 117, p. 21. 156

Jarrett, K., Kavukcuoglu, K., Ranzato, M., & LeCun, Y. (2009) What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pp. 2146–2153. 27

Jing, L., & Tian, Y. (2020) Self-supervised visual feature learning with deep neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* **43** (11): 4037–4058. 155

Johnson, R., & Zhang, T. (2013) Accelerating stochastic gradient descent using predictive variance reduction. *Advances in neural information processing systems* **26**: 315–323. 86

Kanda, N., Takeda, R., & Obuchi, Y. (2013) Elastic spectral distortion for low resource speech recognition with deep neural networks. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pp. 309–314. IEEE. 156

Kang, G., Dong, X., Zheng, L., & Yang, Y. (2017) Patchshuffle regularization. *arXiv preprint arXiv:1707.07103* . 156

Kendall, A., & Gal, Y. (2017) What uncertainties do we need in bayesian deep learning for computer vision? *Advances in neural information processing systems* **30**. 154

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P., (2017) On large-batch training for deep learning: Generalization gap and sharp minima. 155

Keskar, N. S., & Socher, R., (2017) Improving generalization performance by switching from adam to sgd. 88

Kingma, D. P., & Ba, J. (2014) Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* . 87

Kipf, T. N., & Welling, M. (2016) Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308* . 156

Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017) Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems*, pp. 972–981. 28, 105

Koenker, R., & Hallock, K. F. (2001) Quantile regression. *Journal of economic perspectives* **15** (4): 143–156. 66

Kolomiyets, O., Bethard, S., & Moens, M.-F. (2011) Model-portability experiments for textual temporal analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, volume 2, pp. 271–276. ACL; East Stroudsburg, PA. 156

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012) Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, ed. by F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger, volume 25. Curran Associates, Inc. 45, 105, 155, 156

Kukačka, J., Golkov, V., & Cremers, D. (2017) Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686* . 150

Kurenkov, A. (2020) A brief history of neural nets and deep learning. *Skynet Today* . 27

Lakshminarayanan, B., Pritzel, A., & Blundell, C. (2017) Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems* **30**. 154

Lasseck, M. (2018) Acoustic bird detection with deep convolutional neural networks. In *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2018 Workshop (DCASE2018)*, pp. 143–147. 156

Lattimore, T., & Szepesvári, C. (2020) *Bandit algorithms*. Cambridge University Press. 129

Lecun, Y. (1985) Une procedure d'apprentissage pour reseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85, Paris, France*, pp. 599–604. 105

LeCun, Y., Bengio, Y., & Hinton, G. (2015) Deep learning. *nature* **521** (7553): 436–444. 45

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998a) Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86** (11): 2278–2324. 156

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (1998b) *Efficient BackProp*, pp. 9–50. Springer Berlin Heidelberg. 105

Li, C., Chen, C., Carlson, D., & Carin, L. (2016a) Preconditioned stochastic gradient langevin dynamics for deep neural networks. In *Thirtieth AAAI Conference on Artificial Intelligence*. 155

Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2016b) Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR* **abs/1603.06560**. 129

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., & Chintala, S., (2020) Pytorch distributed: Experiences on accelerating data parallel training. 106

Li, Y., Cohn, T., & Baldwin, T. (2017) Robust training under linguistic adversity. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pp. 21–27. 156

Lin, T., Goyal, P., Girshick, R. B., He, K., & Dollár, P. (2017) Focal loss for dense object detection. *CoRR* **abs/1708.02002**. 66, 67

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., & Han, J., (2021a) On the variance of the adaptive learning rate and beyond. 88

Liu, L., Luo, Y., Shen, X., Sun, M., & Li, B. (2019) Beta-dropout: A unified dropout. *IEEE Access* **7**: 36140–36153. 154

Liu, X., Zhang, F., Hou, Z., Mian, L., Wang, Z., Zhang, J., & Tang, J. (2021b) Self-supervised learning: Generative or contrastive. *IEEE Transactions on Knowledge and Data Engineering* . 156

Loshchilov, I., & Hutter, F. (2017) Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* . 88, 151

Louizos, C., Welling, M., & Kingma, D. P., (2018) Learning sparse neural networks through l_0 regularization. 152

Lu, Z., Pu, H., Wang, F., Hu, Z., & Wang, L. (2017) The expressive power of neural networks: A view from the width. *CoRR* **abs/1709.02540**. 45, 46

Ma, Y.-A., Chen, T., & Fox, E. (2015) A complete recipe for stochastic gradient mcmc. *Advances in neural information processing systems* **28**. 155

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013) Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, p. 3. 28

MacKay, D. J., & others. (1995) Ensemble learning and evidence maximization. In *Proc. Nips*, volume 10, p. 4083. Citeseer. 155

Mangalam, K., Fan, H., Li, Y., Wu, C.-Y., Xiong, B., Feichtenhofer, C., & Malik, J. (2022) Reversible vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10830–10840. 169

McCulloch, W. S., & Pitts, W. (1943) A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* **5** (4): 115–133. 27

Mezard, M., & Mora, T., (2008) Constraint satisfaction problems and neural networks: a statistical physics perspective. 87

Min, J., McCoy, R. T., Das, D., Pitler, E., & Linzen, T. (2020) Syntactic data augmentation increases robustness to inference heuristics. *arXiv preprint arXiv:2004.11999* . 156

Minsky, M., & Papert, S. A. (1969) *Perceptrons: An introduction to computational geometry*. MIT press. 27

Montúfar, G. (2017) Notes on the number of linear regions of deep neural networks. 45

Montúfar, G., Pascanu, R., Cho, K., & Bengio, Y. (2014) On the number of linear regions of deep neural networks. *arXiv preprint arXiv:1402.1869* . 45

Moreno-Torres, J. G., Raeder, T., Alaiz-Rodríguez, R., Chawla, N. V., & Herrera, F. (2012) A unifying view on dataset shift in classification. *Pattern recognition* **45** (1): 521–530. 129

Müller, R., Kornblith, S., & Hinton, G. E. (2019) When does label smoothing help? *Advances in neural information processing systems* **32**. 155

Mun, S., Shon, S., Kim, W., Han, D. K., & Ko, H. (2017) Deep neural network based learning and transferring mid-level audio features for acoustic scene classification. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 796–800. 156

Nair, V., & Hinton, G. E. (2010) Rectified linear units improve restricted boltzmann machines. In *Icml*. 27

Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., & Sutskever, I., (2019) Deep double descent: Where bigger models and more data hurt. 123, 127, 128

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., & Zaharia, M., (2021a) Memory-efficient pipeline-parallel dnn training. 106

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., & Zaharia, M., (2021b) Efficient large-scale language model training on gpu clusters using megatron-lm. 107

Neal, R. M. (1995) *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media. 155

Nesterov, Y. E. (1983) A method for solving the convex programming problem with convergence rate o (1/k^ 2). In *Dokl. akad. nauk Sssr*, volume 269, pp. 543–547. 87

Neyshabur, B., Bhojanapalli, S., McAllester, D., & Srebro, N., (2017a) Exploring generalization in deep learning. 127

Neyshabur, B., Bhojanapalli, S., & Srebro, N. (2017b) A pac-bayesian approach to spectrally-normalized margin bounds for neural networks. *arXiv preprint arXiv:1707.09564* . 152

Ng, N. H., Gabriel, R. A., McAuley, J., Elkan, C., & Lipton, Z. C. (2017) Predicting surgery duration with neural heteroscedastic regression. In *Machine Learning for Healthcare Conference*, pp. 100–111. PMLR. 67

Niu, F., Recht, B., Re, C., & Wright, S. J., (2011) Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. 106

Nix, D. A., & Weigend, A. S. (1994) Estimating the mean and variance of the target probability distribution. In *Proceedings of 1994 ieee international conference on neural networks (ICNN'94)*, volume 1, pp. 55–60. IEEE. 66

Noci, L., Roth, K., Bachmann, G., Nowozin, S., & Hofmann, T. (2021) Disentangling the roles of curation, data-augmentation and the prior in the cold posterior effect. *Advances in Neural Information Processing Systems* **34**. 155

Noroozi, M., & Favaro, P. (2016) Unsupervised learning of visual representations by solving jigsaw puzzles. In *European conference on computer vision*, pp. 69–84. Springer. 155

Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., & Le, Q. V. (2019) Specaugment: A simple data augmentation method for automatic speech recognition. *arXiv preprint arXiv:1904.08779* . 156

Parker, D. B. (1985) Learning-logic: Casting the cortex of the human brain in silicon. 105

Pascanu, R., Montufar, G., & Bengio, Y. (2014) On the number of inference regions of deep feed forward networks with piece-wise linear activations. corr. 45

Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., & Efros, A. A. (2016) Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2536–2544. 155

Pereyra, G., Tucker, G., Chorowski, J., Kaiser, ., & Hinton, G., (2017) Regularizing neural networks by penalizing confident output distributions. 155

Poggio, T. A., Mhaskar, H., Rosasco, L., Miranda, B., & Liao, Q. (2016) Why and when can deep - but not shallow - networks avoid the curse of dimensionality: a review. *CoRR* **abs/1611.00740**. 46

Polyak, B. (1964) Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics* **4** (5): 1–17. 87

Prince, S. J. (2012) *Computer vision: models, learning, and inference*. Cambridge University Press. 155

Prokudin, S., Gehler, P., & Nowozin, S., (2018) Deep directional statistics: Pose estimation with uncertainty quantification. 67

Qi, J., Du, J., Siniscalchi, S. M., Ma, X., & Lee, C.-H. (2020) On mean absolute error for deep neural network based vector-to-vector regression. *IEEE Signal Processing Letters* **27**: 1485–1489. 66

Qiu, S., Xu, B., Zhang, J., Wang, Y., Shen, X., De Melo, G., Long, C., & Li, X. (2020) Easyaug: An automatic textual data augmentation platform for classification tasks. In *Companion Proceedings of the Web Conference 2020*, pp. 249–252. 156

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., & others. (2019) Language models are unsupervised multitask learners. *OpenAI blog* **1** (8): 9. 155

Ramachandran, P., Zoph, B., & Le, Q. V., (2017) Searching for activation functions. 28

Reddi, S. J., Kale, S., & Kumar, S. (2018) On the convergence of adam and beyond. In *International Conference on Learning Representations*. 87

Robbins, H., & Monro, S. (1951) A stochastic approximation method. *The Annals of Mathematical Statistics* **22** (3): 400–407. 86

Rodrigues, F., & Pereira, F. C., (2018) Beyond expectation: Deep joint mean and quantile regression for spatio-temporal problems. 66

Rosenblatt, F. (1958) The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* **65** (6): 386. 27

Ruder, S. (2016) An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* . 86

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985) Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science. 105

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986) Learning representations by back-propagating errors. *nature* **323** (6088): 533–536. 105

Safran, I., & Shamir, O. (2016) Depth separation in relu networks for approximating smooth non-linear functions. *CoRR* **abs/1610.09887**. 46

Salamon, J., & Bello, J. P. (2017) Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal processing letters* **24** (3): 279–283. 156

Schneider, S., Baevski, A., Collobert, R., & Auli, M. (2019) wav2vec: Unsupervised pre-training for speech recognition. *arXiv preprint arXiv:1904.05862* . 156

Schwarz, J., Jayakumar, S., Pascanu, R., Latham, P., & Teh, Y. (2021) Powerpropagation: A sparsity inducing weight reparameterisation. *Advances in Neural Information Processing Systems* **34**. 152

Sejnowski, T. J. (2018) *The deep learning revolution*. MIT press. 27

Serra, T., Tjandraatmadja, C., & Ramalingam, S. (2018) Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pp. 4558–4566. PMLR. 45

Shang, W., Sohn, K., Almeida, D., & Lee, H. (2016) Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international* *conference on machine learning*, pp. 2217–2225. 28

Sharif Razavian, A., Azizpour, H., Sullivan, J., & Carlsson, S. (2014) Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 806–813. 155

Shen, X., Tian, X., Liu, T., Xu, F., & Tao, D. (2017) Continuous dropout. *IEEE transactions on neural networks and learning systems* **29** (9): 3926–3937. 154

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B., (2020) Megatron-lm: Training multi-billion parameter language models using model parallelism. 107

Shorten, C., & Khoshgoftaar, T. M. (2019) A survey on image data augmentation for deep learning. *Journal of big data* **6** (1): 1–48. 156

Sjöberg, J., & Ljung, L. (1995) Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control* **62** (6): 1391–1407. 153

Smith, S., Elsen, E., & De, S. (2020) On the generalization benefit of noise in stochastic gradient descent. In *International Conference on Machine Learning*, pp. 9058–9067. PMLR. 153

Smith, S. L., Dherin, B., Barrett, D. G. T., & De, S., (2021) On the origin of implicit regularization in stochastic gradient descent. 153

Snoek, J., Larochelle, H., & Adams, R. P. (2012) Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959. 129

Sohoni, N. S., Aberger, C. R., Leszczynski, M., Zhang, J., & Ré, C., (2019) Low-memory neural network training: A technical report. 106

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014) Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* **15** (1): 1929–1958. 154

Summers, C., & Dinneen, M. J. (2019) Improved mixed-example data augmentation. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1262–1270. IEEE. 156

Sun, R.-Y. (2020) Optimization for deep learning: An overview. *Journal of the Operations Research Society of China* **8** (2): 249–294. 86

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013) On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, ed. by S. Dasgupta & D. McAllester, volume 28 of *Proceedings of Machine Learning Research*, pp. 1139–1147. PMLR. 87

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016) Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826. 150, 155

Telgarsky, M. (2016) Benefits of depth in neural networks. *CoRR* . 46

Vapnik, V. (1995) *The Nature of Statistical Learning Theory.* Springer Verlag. 67

Vapnik, V. N., & Chervonenkis, A. Y. (1971) On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pp. 11–30. Springer. 127

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017) Attention is all you need. *Advances in neural information processing systems* **30**. 155

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., & Fergus, R. (2013) Regularization of neural networks using dropconnect. In *International conference on machine learning*, pp. 1058–1066. PMLR. 154

Wei, J., & Zou, K. (2019) Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196* . 156

Welling, M., & Teh, Y. W. (2011) Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 681–688. Citeseer. 155

Wenzel, F., Roth, K., Veeling, B. S., Świątkowski, J., Tran, L., Mandt, S., Snoek, J., Salimans, T., Jenatton, R., & Nowozin, S. (2020a) How good is the bayes posterior in deep neural networks really? *arXiv preprint arXiv:2002.02405* . 155

Wenzel, F., Snoek, J., Tran, D., & Jenatton, R. (2020b) Hyperparameter ensembles for robustness and uncertainty quantification. *Advances in Neural Information Processing Systems* **33**: 6514–6527. 154

Werbos, P. (1974) Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University* . 105

Williams, P. M. (1996) Using neural networks to model conditional multivariate densities. *Neural computation* **8** (4): 843–854. 66

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., & Recht, B., (2018) The marginal value of adaptive gradient methods in machine learning. 88

Wolpert, D. H. (1992) Stacked generalization. *Neural networks* **5** (2): 241–259. 154

Wu, R., Yan, S., Shan, Y., Dang, Q., & Sun, G., (2015) Deep image: Scaling up image recognition. 150

Xia, F., Liu, T.-Y., Wang, J., Zhang, W., & Li, H. (2008) Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pp. 1192–1199. 67

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018) Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, pp. 5393–5402. PMLR. 106

Xie, L., Wang, J., Wei, Z., Wang, M., & Tian, Q. (2016) Disturblabel: Regularizing cnn on the loss layer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4753–4762. 155

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., & Yu, Y. (2015) Petuum: A new platform for distributed machine learning on big data. *IEEE transactions on Big Data* **1** (2): 49–67. 106

Xu, B., Wang, N., Chen, T., & Li, M. (2015) Empirical evaluation of rectified activations in convolutional network. *CoRR* **abs/1505.00853**. 154, 156

Xu, P., Kumar, D., Yang, W., Zi, W., Tang, K., Huang, C., Cheung, J. C. K., Prince, S. J., & Cao, Y. (2021) Optimizing deeper transformers on small datasets. *arXiv preprint arXiv:2012.15355* . 106

Ye, H., & Young, S. (2004) High quality voice morphing. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pp. I–9. IEEE. 156

Yoshida, Y., & Miyato, T. (2017) Spectral norm regularization for improving the generalizability of deep learning. *arXiv preprint arXiv:1705.10941* . 152

You, Y., Chen, T., Wang, Z., & Shen, Y. (2020) When does self-supervision help graph convolutional networks? In *international conference on machine learning*, pp. 10871–10880. PMLR. 156

Yun, S., Han, D., Oh, S. J., Chun, S., Choe, J., & Yoo, Y. (2019) Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 6023–6032. 156

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018) Adaptive methods for nonconvex optimization. *Advances in neural information processing systems* **31**. 88

Zaslavsky, T. (1975) *Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes: Face-count formulas for partitions of space by hyperplanes*, volume 154. American Mathematical Soc. 28, 31

Zeiler, M. D. (2012) Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* . 87

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016a) Understanding deep learning requires rethinking generalization. *CoRR* **abs/1611.03530**. 152

Zhang, H., Cisse, M., Dauphin, Y. N., & Lopez-Paz, D. (2017) mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412* . 156

Zhang, H., Dauphin, Y. N., & Ma, T., (2019) Fixup initialization: Residual learning without normalization. 106

Zhang, H., Hsieh, C.-J., & Akella, V. (2016b) Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638. 106

Zhang, R., Isola, P., & Efros, A. A. (2016c) Colorful image colorization. In *European conference on computer vision*, pp. 649–666. Springer. 155

Zhong, Z., Zheng, L., Kang, G., Li, S., & Yang, Y. (2020) Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 13001–13008. 156

Zhu, C., Ni, R., Xu, Z., Kong, K., Huang, W. R., & Goldstein, T. (2021) Gradinit: Learning to initialize neural networks for stable and efficient training. *Advances in Neural Information Processing Systems* **34**. 106

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., & He, Q. (2020) A comprehensive survey on transfer learning. *Proceedings of the IEEE* **109** (1): 43–76. 155

Index