CS 230: Deep Learning

Hargen Zheng

December 26, 2023

Contents

Part I

Neural Networks and Deep Learning

Chapter 1

Introduction

Why it matters?

- AI is the new Electricity.
- Electricity had once transformed countless industries: transportation, manufacturing, healthcare, communications, and more.
- AI will now bring about an equally big transformation.

Courses in this sequence (Specialization):

1. Neural Networks and Deep Learning
2. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization
3. Structuring your Machine Learning project
4. Convolutional Neural Networks (CNNs)
5. Natural Language Processing: Building sequence models (RNNs, LSTM)

1.1 What is a Neural Network?

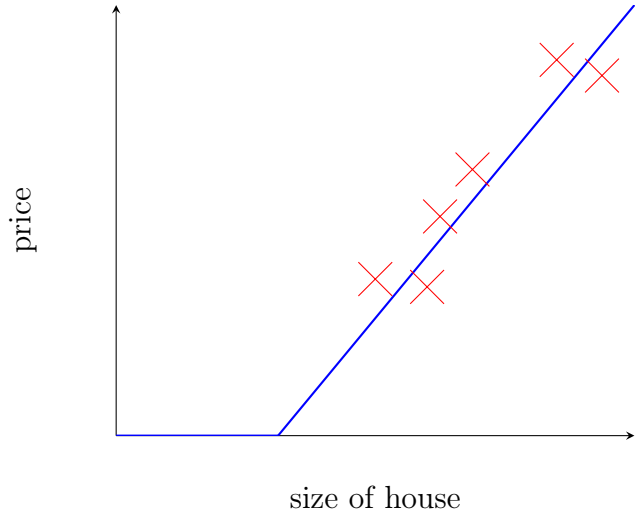

Figure 1.1: Housing Price Prediction

In the neural network literature, this function appears by a lot. This function is called a ReLU function, which stands for rectified linear units.

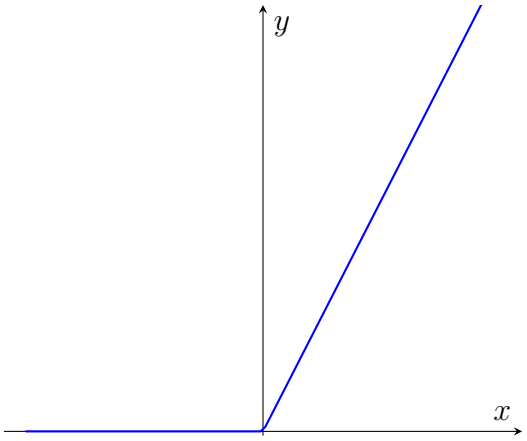

Figure 1.2: ReLU function: $\max\{0, x\}$

With the size of houses in square feet or square meters and the price of the house, we can fit a function to predict the price of a house as a function of its size.

This is the simplest neural network. We have the size of a house as input x, which goes into a node (a single "neuron"), and outputs the price y.

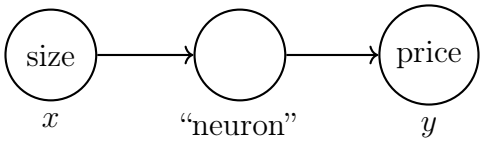

Figure 1.3: Simple Neural Network of Housing Example

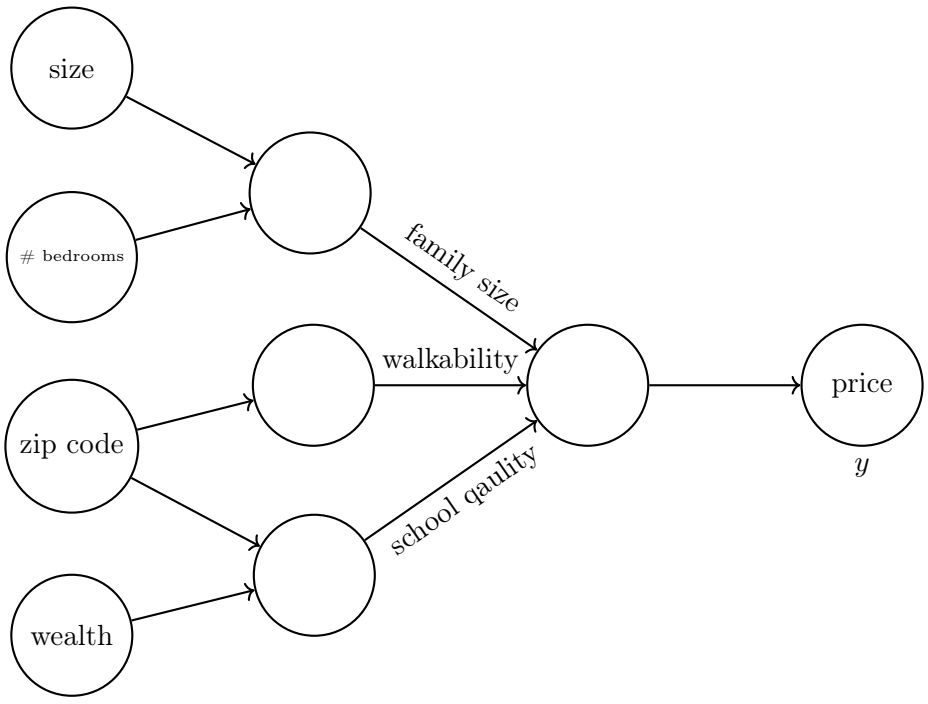

Figure 1.4: Slightly Larger Neural Network

Suppose that, instead of predicting the price of a house just from the size, we also have other features, such as the number of bedrooms, zip code, and wealth. The number of bedrooms determines whether or not a house can fit one's family size; Zip code tells walkability; and zip code and wealth tells how good the school quality is. Each of the circle could be ReLU or other nonlinear function.

We need to give the neural network the input x and the output y for a number of examples in the training set and, for all the things in the middle, neural network will figure out by itself.

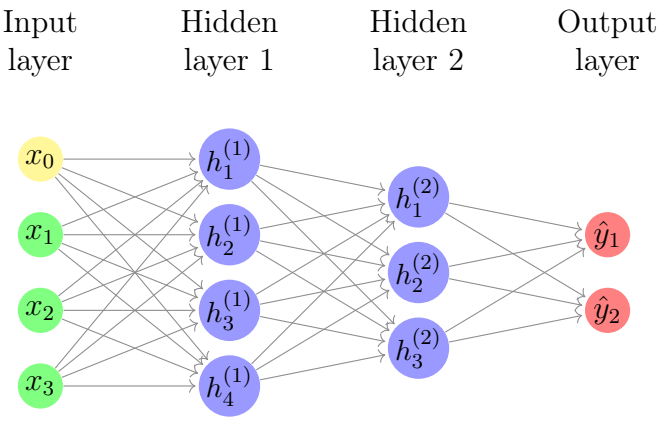

Figure 1.5: General Picture of Neural Network

1.2 Supervised Learning with Neural Networks

Supervised Learning Examples.

Input(x)	Output(y)	Application	Type of NN
Home features	Price	Real Estate	Standard
Ad, user info	Click on ad? (0/1)	Online Advertising	Standard
Image	Object $(1, \ldots, 1000)$	Photo tagging	CNN
Audio	Text transcript	Speech recognition	RNN
English	Chinese	Machine translation	RNN
Image, Radar info	Position of other cars	Autonomous driving	Custom

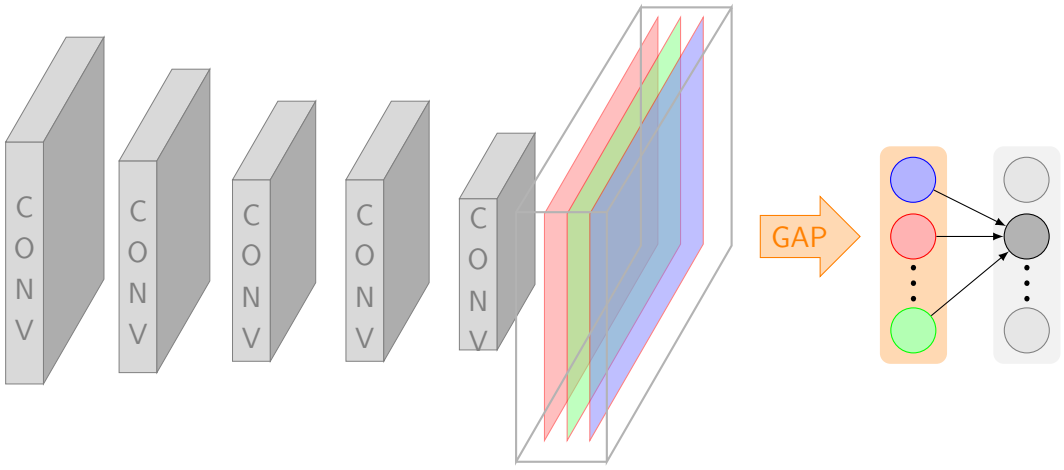

Figure 1.6: Convolutional Neural Network

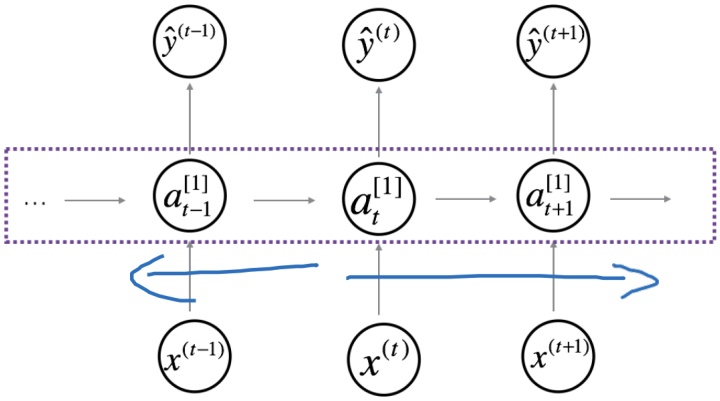

Figure 1.7: Recurrent Neural Network

There are two types of data: Structured Data and Unstructured Data. We could have the following examples for each.

Size	#bedrooms	···	Price(1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad ID	···	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Table 1.1: Structured Data

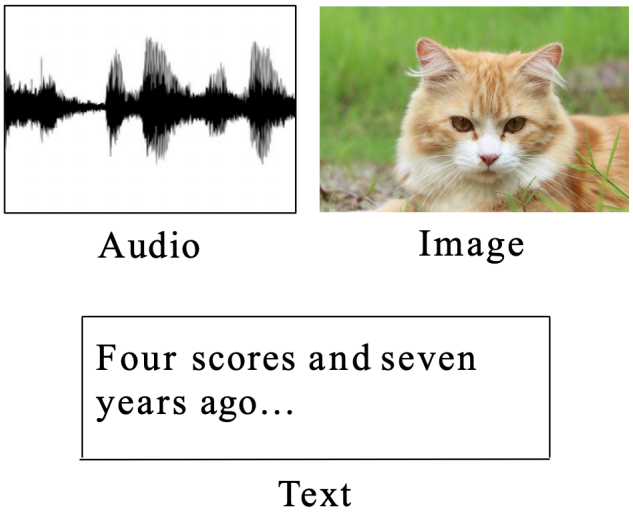

Figure 1.8: Unstructured Data

Historically, it has been much harder for computers to make sense of unstructured data compared to structured data. Thanks to the neural networks, computers are better at interpreting unstructured data as well, compared to just a few years ago.

1.3 Why is Deep Learning Taking off?

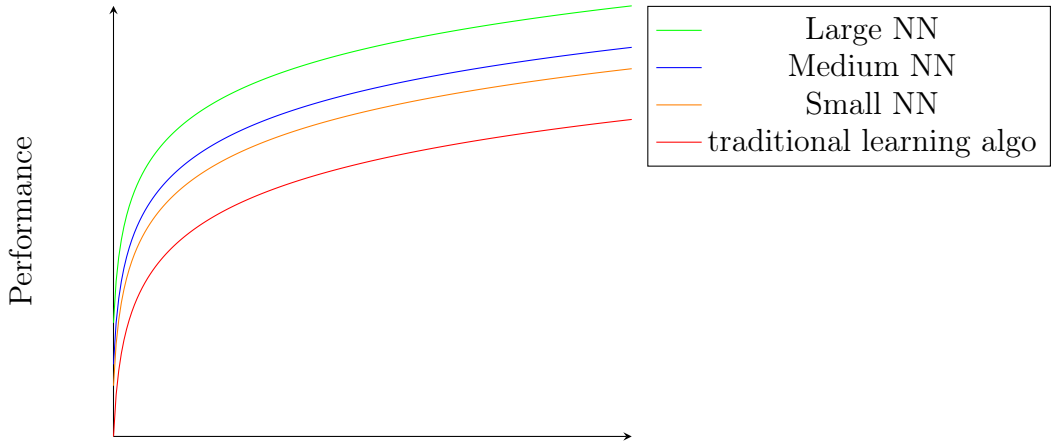

Figure 1.9: Scale drives deep learning progress

In the regime of small number of training sets, the relative ordering of the algorithms is not very well defined. If you don't have a lot of training data, it is often up to your skill at hand engineering features that determines performance. When we have large training sets – large labeled data regime in the right, we more consistently see large neural networks dominating the other approaches.

Chapter 2

Neural Networks Basics

2.1 Logistic Regression as a Neural Network

When implementing a neural network, you usually want to process entire training set without using an explicit for-loop. Also, when organizing the computation of a neural network, usually you have what's called a forward pass or forward propagation step, followed by a backward pass or backward propagation step.

2.1.1 Notation

Each training set will be comprised of m training examples:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}.$$

We denote m_{train} to be the number of training examples in the training set, and m_{test} to be the number of test examples. The ith training example is represented by a pair $(x^{(i)}, y^{(i)})$, where $x^{(i)} \in \mathbb{R}^{n_x}$ and $y^{(i)} \in \{0, 1\}$. To put all of the training examples in a more compact notation, we define matrix X as

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix}$$

where $X \in \mathbb{R}^{n_x \times m}$. Similarly, we define Y as

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

where $Y \in \mathbb{R}^{1 \times m}$.

2.1.2 Binary Classification

Figure 2.1: Cat Image

You might have an input of an image and want to output a label to recognize this image as either being a cat, in which case you output 1, or non-cat, in which case you output 0. We use y to denote the output label.

To store an image, computer stores three separate matrices corresponding to the red, green, and blue color channels of this image. If the input image is 64 pixels by 64 pixels, then you would have 3 64×64 matrices corresponding to the red, green, and blue pixel intensity values for the image. We could define a feature vector x as follows:

$$x = \begin{bmatrix} | & | & | \\ red & green & blue \\ | & | & | \end{bmatrix}.$$

Suppose each matrix has dimension 64×64, then

$$n_x = 64 \times 64 \times 3 = 12288.$$

2.1.3 Logistic Regression

Given an input feature $x \in \mathbb{R}^{n_x}$, we want to find a prediction $\hat{y}$, where we want to interpret $\hat{y}$ as the probability of $y = 1$ for a given set of input features x.

$$\hat{y} = P(y = 1|x).$$

Define the weights $w \in \mathbb{R}^{n_x}$ and bias term $b \in \mathbb{R}$ in the logistic regression. It turns out, when implementing the neural networks, it is better to keep parameters w and b separate, instead of putting them together as $\theta \in \mathbb{R}^{n_x+1}$.

In linear regression, we would use $\hat{y} = w^T x + b$. However, this is not good for binary classification because the prediction could be much bigger than 1 or even negative, which does not make sense for probability. Therefore, we apply the sigmoid function, where

$$\hat{y} = \sigma(w^T x + b) = \sigma(z).$$

The formula of the sigmoid function is given as follows:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

If z is very large, then e^{-z} will be close to zero, so $\sigma(z) \approx 1$. Conversely, if z is very small or a very large negative number, then $\sigma(z) \approx 0$.

The graph of the sigmoid function is given below:

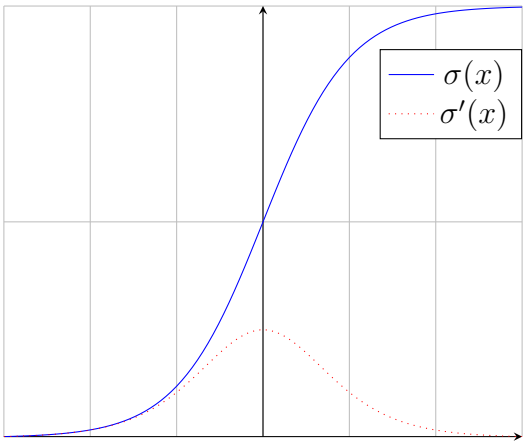

Figure 2.2: Sigmoid Function

2.1.4 Logistic Regression Cost Function

Logistic regression is a supervised learning algorithm, where given training examples $\{(x^{(1)}, y^{(1)}, \ldots, (x^{(m)}, y^{(m)})\}$, we want to find parameters w and b such that $\hat{y}^{(i)} \approx y^{(i)}$.

MSE is not used as the loss function because when learning parameters, the optimization problem becomes non-convex, where we end up with multiple local optima, so gradient descent may not find a global optimum. The loss function is defined to measure how good our output $\hat{y}$ is when the true label is y. Instead, the loss (error) function for logistic regression could be represented by

$$\mathcal{L}(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})).$$

The intuition behind is that when considering the squared error cost function, we want the squared error to be as small as possible. Similarly, we want to loss function for logistic regression to be as small as possible.

If $y = 1$, then the loss function ends up with

$$\mathcal{L}(\hat{y}, 1) = -\log(\hat{y}).$$

In this case, we want $\log(\hat{y})$ to be large $\Leftrightarrow$ want $\hat{y}$ to be large.

If $y = 0$, then

$$\mathcal{L}(\hat{y}, 0) = -\log(1 - \hat{y}).$$

In this case, we want $\log(1 - \hat{y})$ to be large $\Leftrightarrow$ want $\hat{y}$ to be small.

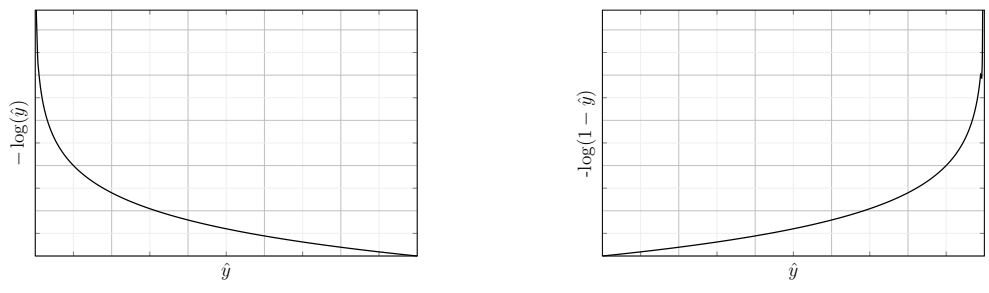

Figure 2.3: A binary classifier's loss function based on if $y = 1$ or $y = 0$.

The loss function is defined with respect to a single training example. The cost function measures how well the model is doing on the entire training set, which is given as

$$\begin{aligned} J(w,b) &= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})). \end{aligned}$$

Interpretation of the Cost Function. We interpret $\hat{y}$ to be the probability of $y = 1$ given a set of input features x. This means that

- If $y = 1$, $P(y|x) = \hat{y}$
- If $y = 0$, $P(y|x) = 1 - \hat{y}$

We can summarize the two equations as follows:

$$P(y|x) = \hat{y}^{y}(1-\hat{y})^{1-y}.$$

Since the log function is a strictly monotonically increasing function, maximizing $\log(P(y|x))$ would give a similar result as maximizing $P(y|x)$.

$$\begin{aligned} \log(P(y|x)) &= \log(\hat{y}^{y}(1-\hat{y})^{1-y}) \\ &= y\log(\hat{y}) + (1-y)\log(1-\hat{y}) \\ &= -\mathcal{L}(\hat{y}, y). \end{aligned}$$

Therefore, minimizing the loss means maximizing the probability.

The above was for one training example. For m examples, we can express the probability as follows:

$$\begin{aligned} \log(P(\text{labels in training set})) &= \log(\prod_{i=1}^{m} P(y^{(i)}|x^{(i)})) \\ &= \sum_{i=1}^{m}\log(P(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \end{aligned}$$

The cost is defined as

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

We get rid of the negative sign as now we want to minimize the cost. $\frac{1}{m}$ is added in the front to make sure our quantities are at a better scale. To minimization of the cost function carries out maximum likelihood estimation with the logistic regression model, under the assumption that our training examples are i.i.d (independent and identically distributed).

2.1.5 Gradient Descent

We want to find w, b that minimize $J(w, b)$. Our cost function $J(w, b)$ is a single big bowl, which is a convex function, which is one of the huge reasons why we use this particular cost function J for logistic regression.

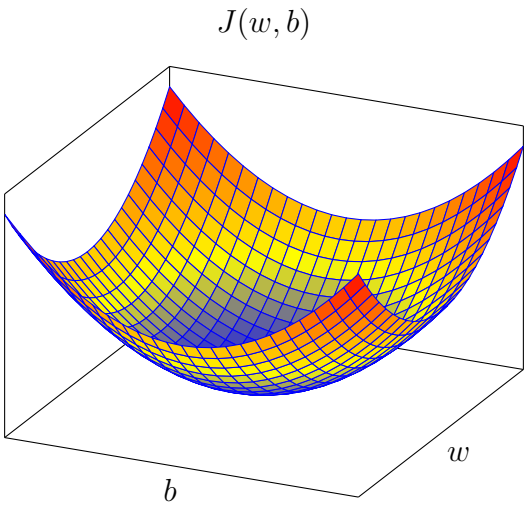

Figure 2.4: Convex Cost Function

For logistic regression, almost any initialization method works. Though random initialization works, people don't usually do that for logistic regression. Instead, we generally initialize the weights to 0. Gradient descent takes a step in the steepest downhill direction. After iterations, we converge to the global optimum or get close to the global optimum. In one dimension, for the parameter w, we can define the gradient descent algorithm as follows

$$w := w - \alpha \frac{\partial J(w, b)}{\partial w}$$
$$b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$

where α is the learning rate that control the size of steps and $\frac{\partial J(w)}{\partial w}, \frac{\partial J(w,b)}{\partial b}$ are the changes we make for the parameter w, b.

2.1.6 Computation Graph

The computations of neural network are organized in terms of a forward pass step, in which we compute the output of the neural network, followed by a backward propagation step, which we use to compute the gradients. The computation graph explains why it is organized this way.

Suppose we want the compute the function

$$J(a, b, c) = 3(a + bc).$$

We can break down the computation into three steps:

1. $u = bc$
2. $v = a + u$
3. $J = 3v$

This could be represented by the computation graph below:

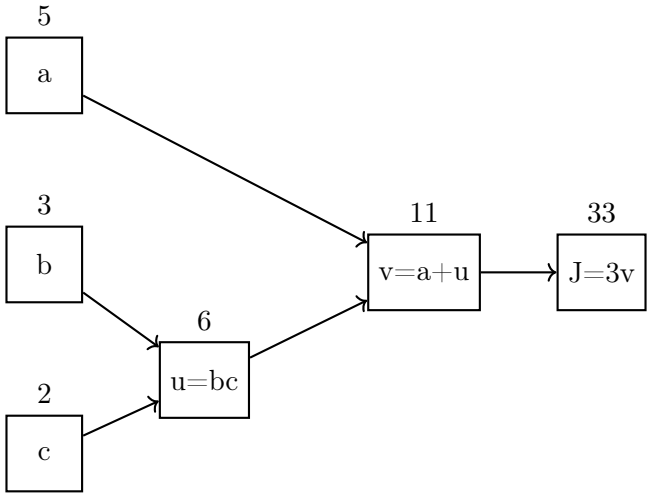

Figure 2.5: Computation Graph for $J(a, b, c) = 3(a + bc)$

Suppose $a = 5, b = 3, c = 2$, then $J(a, b, c) = 3(a + bc) = 3(5 + 3 \times 2) = 33$.

2.1.7 Derivatives with a Computation Graph

Consider the computation graph 2.5. Suppose we want to compute the derivative of J with respect to v. Since we know $J = 3v$, we can take the derivative as follows:

$$\frac{\partial J}{\partial v} = \frac{\partial}{\partial v}(3v) = 3.$$

Suppose then we want to compute the derivative of J with respect to a. We can apply the chain rule as follows:

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a} = 3 \cdot \frac{\partial}{\partial a}(a + u) = 3 \cdot 1 = 3.$$

For brevity, we could use "da'' to represent $\frac{\partial J}{\partial a}$. Similarly, we can use "dv'' to represent $\frac{\partial J}{\partial v}$.

To find the derivative of J with respect to u, we could do similar computation as $\frac{\partial J}{\partial a}$ as follows:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u} = 3 \cdot 1 = 3.$$

$\frac{\partial J}{\partial b}$ could be compute as follows:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial b} = 3 \cdot c = 3 \cdot 2 = 6.$$

Similarly, we can compute

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial c} = 3 \cdot b = 3 \cdot 3 = 9.$$

The most efficient way to compute all these derivatives is through a right-to-left computation following the direction of the arrows below:

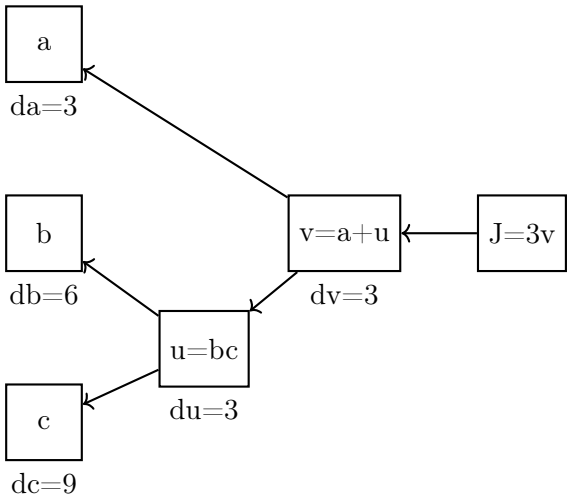

Figure 2.6: Backward Propagation for $J(a, b, c) = 3(a + bc)$

2.1.8 Logistic Regression Gradient Descent

This section will cover how to compute derivatives to implement gradient descent for logistic regression.

To recap, logistic regression is set up as follows:

$$\begin{aligned} z &= w^T x + b \\ \hat{y} &= a = \sigma(z) \\ \mathcal{L}(a, y) &= -(y \log(a) + (1 - y) \log(1 - a)) \end{aligned}$$

This can be represented by the computation graph below:

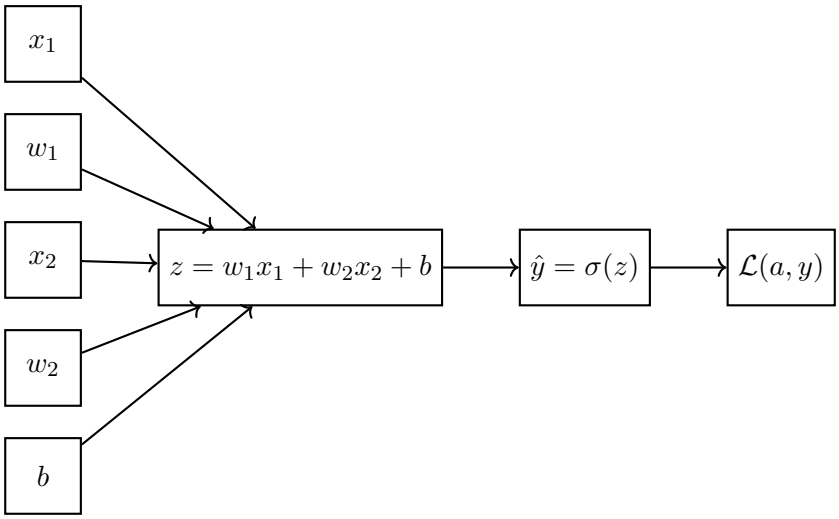

Figure 2.7: Computation Graph for Logistic Regression

In this case, we can compute "da" as follows:

$$da = \frac{\partial \mathcal{L}(a,y)}{\partial a} = -y \cdot \frac{1}{a} - (1-y) \cdot \frac{1}{1-a} \cdot (-1) = -\frac{y}{a} + \frac{1-y}{1-a}.$$

We can give both terms the same denominator and clean up as follows:

$$\frac{\partial \mathcal{L}(a,y)}{\partial a} = \frac{-y(1-a)}{a(1-a)} + \frac{a(1-y)}{a(1-a)} = \frac{-y + ay + a - ay}{a(1-a)} = \frac{a-y}{a(1-a)}.$$

Then, we can go backwards and compute $\frac{\partial \mathcal{L}(a,y)}{\partial z}$. We know that the derivative of a sigmoid function has the form $\frac{\partial}{\partial z}\sigma(z) = \sigma(z)(1-\sigma(z))$. In this case $\sigma(z) = a$, as we have defined, so $\frac{\partial a}{\partial z} = a(1-a)$. Therefore,

$$dz = \frac{\partial \mathcal{L}(a,y)}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} = \frac{a-y}{a(1-a)} \times a(1-a) = a - y.$$

Finally, we do the backward propagation for the input layer as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}(a,y)}{\partial w_1} &= x_1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = x_1 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial w_2} &= x_2 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = x_2 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial x_1} &= w_1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = w_1 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial x_2} &= w_2 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = w_2 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial b} &= 1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = (a-y)
\end{aligned}$$

2.1.9 Gradient Descent on m examples

Recall that the cost function for m training examples is given by

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y),$$

where $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$.

The cost function is the average of loss values for each training example. It turns out that derivative works similarly and we can compute the derivative of $J(w,b)$ with respect to w_1 as follows:

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_i} \mathcal{L}(a^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} x_1 \cdot (a^{(i)} - y^{(i)}).$$

Now, let's formalize the algorithm for the logistic regression gradient descent.

To implement logistic regression this way, we have two for-loops. The first for-loop is to iterate through all training examples. The second for-loop is to iterate through all the features. In the example above, we only have features $w^{(1)}, w^{(2)}$. If we have n features instead, then we need to iterate through all of them.

However, when implementing deep learning algorithms, the explicit for-loop makes the algorithms less efficient. The vectorization technique allows us to get rid of the explicit for-loops, which allows us to scale to larger datasets.

Algorithm 1 Gradient Descent for Logistic Regression

Initialize $J = 0; dw_1 = 0; dw_2 = 0; db = 0$
for $i = 1$ to m **do**
  $z^{(i)} \leftarrow w^T x^{(i)} + b$
  $a^{(i)} \leftarrow \sigma(z^{(i)})$
  $J \leftarrow J + [-(y^{(i)} \log(a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))]$
  $dz^{(i)} \leftarrow a^{(i)} - y^{(i)}$ ▷ Superscript refers to one training example
  $dw_1 \leftarrow dw_1 + x_1^{(i)} dz^{(i)}$ ▷ Here we assume $n_x = 2$
  $dw_2 \leftarrow dw_2 + x_2^{(i)} dz^{(i)}$ ▷ No superscript i as it's accumulative
  $db \leftarrow db + dz^{(i)}$
end for
$J \leftarrow J/m$
$dw_1 \leftarrow dw_1/m; dw_2 \leftarrow dw_2/m; db \leftarrow db/m$

2.2 Python and Vectorization

2.2.1 Vectorization

The aim of vectorization is to remove explicit for-loops in code. In the deep learning era, we often train on relatively large dataset because that's when deep learning algorithms tend to shine. In deep learning, the ability to perform vectorization has become a key skills.

Recall that in logistic regression, we compute $z = w^T x + b$, where $w, x \in \mathbb{R}^{n_x}$. For non-vectorized implementation, we compute z as

$$z = \sum_{i=1}^{n_x} w[i] \times x[i], ; z+ = b$$

The vectorized implementation is very fast:

$$z = np.dot(w, x) + b.$$

SIMD (single instruction multiple data), in both GPU and CPU, enables built-in functions, such as $np.dot()$, to take much better advantage of parallelism to do computations much faster.

Neural network programming guideline: whenever possible, avoid explicit for-loops.

If we want to compute $u = Av$, then by the definition of matrix multiplication,

$$u_i = \sum_j A_{ij} v_j.$$

The vectorized implementation would be

$$u = np.dot(A, v).$$

Suppose we want to apply the exponential operation on every element of a vector

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

We can apply the vectorization technique as follows:

$$u = np.exp(v).$$

Similarly, we have $np.log(v), np.abs(v), np.maximum(v, 0), v**2$, etc. to apply the vectorization technique. We can revise the original gradient descent algorithm and use the vectorization technique as follows

Algorithm 2 Gradient Descent for Logistic Regression with Vectorization

Initialize $J = 0; dw = np.zeros((n_x, 1)); db = 0$
for $i = 1$ to m **do**
  $z^{(i)} \leftarrow w^T x^{(i)} + b$
  $a^{(i)} \leftarrow \sigma(z^{(i)})$
  $J \leftarrow J + [-(y^{(i)} \log(a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))]$
  $dz^{(i)} \leftarrow a^{(i)} - y^{(i)}$
  $dw \leftarrow dw + x^{(i)} dz^{(i)}$ ▷ Get rid of the for-loop
  $db \leftarrow db + dz^{(i)}$
end for
$J \leftarrow J/m$
$dw \leftarrow dw/m; db \leftarrow db/m$

2.2.2 Vectorizing Logistic Regression

For logistic regression, we need to compute $a^{(i)} = \sigma(z^{(i)})$ m times, where $z^{(i)} = w^T x^{(i)} + b$. There is a way to compute without using explicit for-loops. Recall that we define

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix},$$

where $X \in \mathbb{R}^{n_x \times m}$.

We want to compute all $z^{(i)}$'s at the same time.

From math, we know that

$$\begin{aligned} Z &= \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} \\ &= w^T X + \begin{bmatrix} b & b & \cdots & b \end{bmatrix} \\ &= \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & \cdots & w^T x^{(m)} + b \end{bmatrix} \end{aligned}$$

In order to implement this in NumPy, we can do

$$Z = np.dot(w.T, X) + b,$$

where $b \in \mathbb{R}$. Python will expand this real number to a $1 \times m$ vector by **broadcasting**.

Similarly, we can stack lower case $a^{(i)}$'s horizontally to obtain capital A as follows:

$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \sigma(Z).$$

With Z being a $1 \times m$ vector, we can apply the sigmoid function to each element of Z by

$$A = \frac{1}{1 + np.exp(-Z)}.$$

2.2.3 Vectorizing Logistic Regression's Gradient Computation

Recall that we compute individual dz's as $dz^{(i)} = a^{(i)} - y^{(i)}$ for the ith training example. Now, let's define

$$dZ = \begin{bmatrix} dz^{(1)} & dz^{(2)} & \cdots & dz^{(m)} \end{bmatrix},$$

where $dz^{(i)}$'s are stacked horizontally and $dZ \in \mathbb{R}^{1\times m}$.

Since we have

$$A = \begin{bmatrix} a^{(1)} & \cdots & a^{(m)} \end{bmatrix}, \; Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(m)} \end{bmatrix}$$

we can compute dZ as

$$dZ = A - Y = \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & \cdots & a^{(m)} - y^{(m)} \end{bmatrix}.$$

We can compute db with vectorization as

$$\begin{aligned} db &= \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} \\ &= \frac{1}{m} \times np.sum(dZ). \end{aligned}$$

Similarly, we can compute dw with vectorization as

$$\begin{aligned} dw &= \frac{1}{m} X dZ^T \\ &= \frac{1}{m} \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix} \\ &= \frac{1}{m} \begin{bmatrix} x^{(1)} dz^{(1)} + \cdots + x^{(m)} z^{(m)} \end{bmatrix}, \end{aligned}$$

where $dw \in \mathbb{R}^{n_x \times 1}$.

2.2.4 Broadcasting in Python

The following matrix shows the calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

Suppose we want to calculate the percentage of calories from Carb, Protein, and Fat for each of the four foods. For example, for apples, the percentage of calories for Carb is

$$\frac{56.0}{59.0} \approx 94.9\%.$$

We could do this without explicit for-loops. Suppose the matrix is denoted by A, where $A \in \mathbb{R}^{3\times 4}$.

To sum each column vertically, we can use

$$cal = A.sum(axis = 0).$$

$axis = 0$ refers to the vertical columns inside the matrix, whereas $axis = 1$ will sum horizontally. The $cal = cal.reshape(1, 4)$ operation would convert s to a one by four vector, which is very cheap to call as it's runtime is $\mathcal{O}(1)$.

Then, we can compute percentages by

$$percentage = 100 * A/(cal.reshape(1, 4)),$$

where we divide a $(3, 4)$ matrix by a $(1, 4)$ vector. This will divide each of the three elements in a column by the value in the corresponding column of $cal.reshape(1, 4)$.

A few more examples follows are provided below. Suppose we want to add 100 to every element of the vector $\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$. We can do

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100.$$

Python will convert the number 100 into the vector

$$\begin{bmatrix}100\\100\\100\\100\end{bmatrix}$$

and so

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 \Rightarrow \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}.$$

If we add a (m, n) matrix by a $(1, n)$ vector, Python will copy the vector m times to convert it into an (m, n) matrix, and then add two matrices together.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}.$$

Now, suppose we add a (m, n) matrix by a $(m, 1)$ vector. Then, Python will copy the vector n times horizontally and form a (m, n) matrix.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}.$$

2.2.5 A Note on Python/NumPy Vectors

Let's set

$$a = np.random.randn(5),$$

which creates five random Gaussian variables stored in array a. It turns out if we do $print(a.shape)$, we get $(5,)$. This is called a **rank one array**, which is neither a row vector nor a column vector. In this case, $a.T$ and a look the same. When we call $np.dot(a, a^T)$, we will get a single number, instead of a matrix generated by the outer product.

When doing neural network computations, it is recommended not to use rank one array. Instead, we can create a $(5, 1)$ vector as follows:

$$a = np.random.randn(5, 1),$$

which generates a column vector. Now, if we call $a.T$, we will get a row vector. This way, the behaviors of the vector are easier to understand.

When dealing with a matrix with unknown dimensions, we can use an assertion statement to make sure the shape is as desired:

$$assert(a.shape == (5, 1)).$$

Also, if we ultimately encounter the rank one array, we can always reshape it to the column vector as follows:

$$a = a.reshape((5, 1)),$$

so it behaves more consistently as a column vector, or a row vector.

Chapter 3

Shallow Neural Networks

3.1 Neural Networks Overview

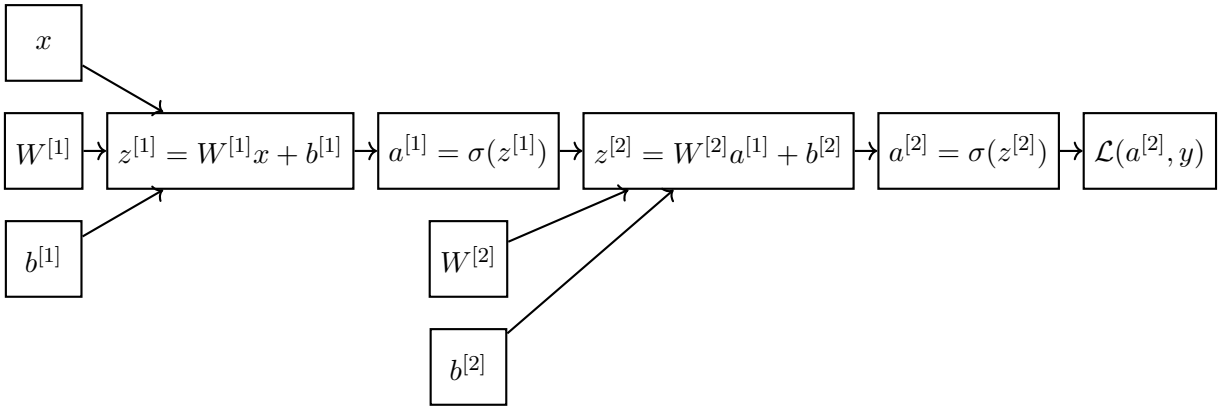

Figure 3.1: Shallow Neural Network

After forward pass, we do backward propagation to compute the derivatives

$$da^{[2]}, dz^{[2]}, dW^{[2]}, db^{[2]}, da^{[1]}, dz^{[1]}, dW^{[1]}, db^{[1]}, dx.$$

This is basically taking logistic regression and repeating it twice.

3.2 Neural Network Representation

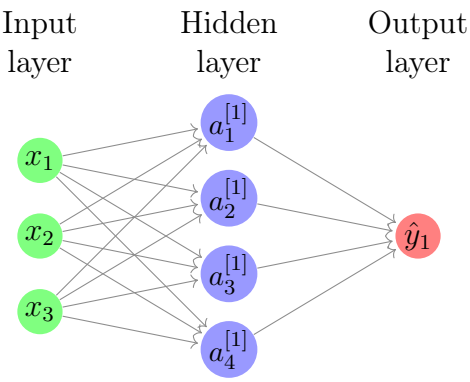

Figure 3.2: Neural Network Representation

In a neural network that we train with supervised learning, the training set contains values of the inputs x as well as the target outputs y. The term "Hidden layer" refers to the fact that in the training set, the true values for those nodes in the middle are not observed.

Alternatively, we can use $a^{[0]}$, which stands for the activation, to represent the input layer X. Then, $a^{[1]}$ can be used to represent the hidden layer, where

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}.$$

Finally, the output layer could be denoted by $a^{[2]}$.

The figure 3.2 is called 2 layer Neural Network. The reason is that when we count layers in neural networks, we do not count the input layer, so hidden layer is layer 1 and output layer is layer 2. Input layer is referred to as layer 0. For the hidden layer, there are parameters $w^{[1]}, b^{[1]}$ associated with it, where $w^{[1]} \in \mathbb{R}^{4\times 3}$ and $b^{[1]} \in \mathbb{R}^{4\times 1}$. Similarly, the output layer has parameters $w^{[2]}, b^{[2]}$ associated with it, where $w^{[2]} \in \mathbb{R}^{1\times 4}$ and $b^{[2]} \in \mathbb{R}^{1\times 1}$.

3.3 Computing a Neural Network's Output

Each node in the hidden layer involves two-step computation:

- $z_j^{[i]} = w_j^{[i]T} x + b_j^{[i]}$, where i represent the index of layer and j represent the node index in that layer.
- $a_j^{[i]} = \sigma(z_j^{[i]})$.

If we refer to 3.2, then we can represent the hidden units with the following equations:

$$\begin{aligned}
z_1^{[1]} &= w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]}) \\
z_2^{[1]} &= w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]}) \\
z_3^{[1]} &= w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]}) \\
z_4^{[1]} &= w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})
\end{aligned}$$

Using for-loop to compute these equations would be pretty inefficient. Therefore, we implement vectorization technique.

As each hidden unit has a corresponding parameter vector w, we can stack these parameter vectors together to form a matrix. Multiplying the resulting matrix with vector x and adding the whole multiplication result with vector b would give us the following:

$$z^{[1]} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}.$$

The matrix that contains the parameter vector could be referred to as $W^{[1]}$ and the bias vector could be referred to as $b^{[1]}$ to brevity, so $z^{[1]} = W^{[1]} x + b^{[1]}$.

Similarly, we could use vectorization to compute $a^{[1]}$ as follows:

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]}).$$

In summary, given input x, or $a^{[0]}$, we perform the following computations:

- $z^{[1]} = W^{[1]}x + b^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$, where $z^{[1]}$ has shape $(4, 1)$, $W^{[1]}$ has shape $(4, 3)$, $a^{[0]}$ has shape $(3, 1)$, and $b^{[1]}$ has shape $(4, 1)$.
- $a^{[1]} = \sigma(z^{[1]})$, where $a^{[1]}$ has shape $(4, 1)$ and $z^{[1]}$ has shape $(4, 1)$.
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$, where $z^{[2]}$ has shape $(1, 1)$, $W^{[2]}$ has shape $(1, 4)$, $a^{[1]}$ has shape $(4, 1)$ and $b^{[2]}$ has shape $(1, 1)$.
- $\hat{y} = a^{[2]} = \sigma(z^{[12]})$, where both $a^{[2]}$ and $z^{[2]}$ have shape $(1, 1)$.

3.4 Vectorizing Across Multiple Examples

Previously, we have only computed the value of $\hat{y}$ for one single input x. For m training examples, we would need to run a for-loop to compute $a^{[2](i)}$, where (i) represents the ith training example and [2] represents layer 2, for each and every training example. This is achieved as follows:

Algorithm 3 Forward Propagation with m Training Examples

for $i = 1$ to m **do**
$z^{[1](i)} = W^{[1]}a^{[0](i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$
end for

We would like to vectorize the whole computation to get rid of the for-loop.
Recall that we can stack training examples together for form a matrix as follows:

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix},$$

where X has shape (n_x, m).

It turns out we need to do the following computations with Python broadcasting technique:

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} = W^{[1]}A^{[0]} + b^{[1]} \\ A^{[1]} &= \sigma(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(Z^{[2]}) \end{aligned}$$

Similar to how matrix X was formed, we can stack individual $z^{[1](i)}$'s and $a^{[1](i)}$ horizontally to form matrix $Z^{[1]}$ and $A^{[1]}$ as follows:

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix}, \; A^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ | & | & & | \end{bmatrix}.$$

Note that horizontal indices correspond to different training examples and vertical indices correspond to the activation of different hidden units in the neural network.

3.5 Activation Functions

In previous illustrations, we use **sigmoid function**

$$a = \sigma(z) = \frac{1}{1 + e^{-z}},$$

as an activation function, where $\sigma(z) \in (0, 1)$.

The sigmoid function can be graphed as follows:

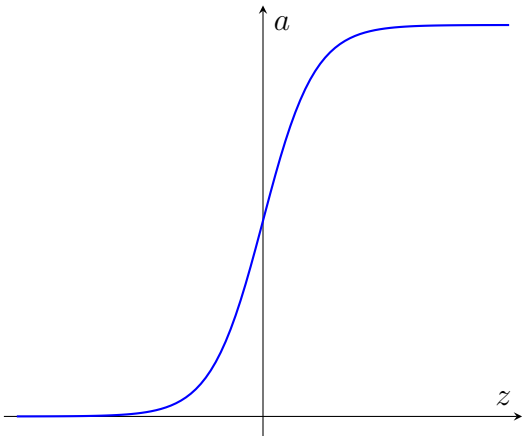

Figure 3.3: Sigmoid Function

More generally, we can use a different activation function $g(z)$ in place of $\sigma(z)$, where $g(z)$ could be a nonlinear function.

Similar to the sigmoid function, we can use **tanh, or hyperbolic tangent**, function

$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

where $a \in (-1, 1)$.

The tanh function can be graphed as follows:

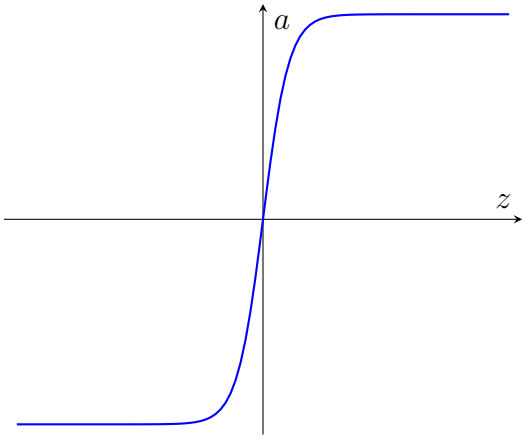

Figure 3.4: Hyperbolic Tangent Function

It turns out that for hidden units, if we let $g(z) = \tanh(z)$, this almost always works better than the sigmoid function. With the values between $+1$ and -1, the mean of the activations that come out of the hidden layer are closer to zero. The tanh function is almost always superior than the sigmoid function. One exception is for the output layer because if y is either zero or one, then it makes sense for $\hat{y}$ to have

$$0 \leq \hat{y} \leq 1.$$

One of the downsides of both tanh and sigmoid functions is that if z is either very large or very small, then the gradient of the function ends up being close to zero, so this can slow down gradient descent. One other choice of activation function, which is very popular in machine learning, is the rectified linear unit (ReLU) function

$$a = \max\{0, z\}.$$

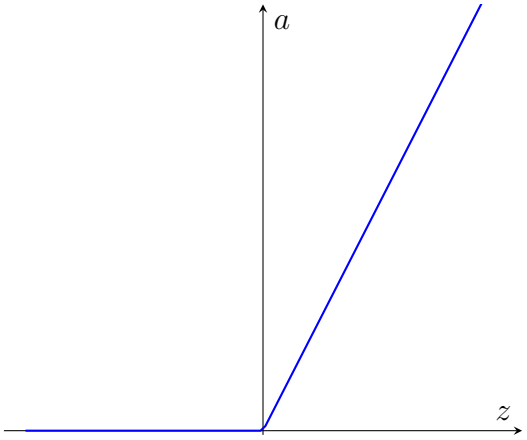

Figure 3.5: ReLU Function

In this case, the derivative is 1 so long as z is positive and the derivative is 0 when z is negative. Technically, when $z = 0$, the derivative is not well defined. However, when we implement this in the computer, the odds that we get exactly $z = 0.00000000000$ is very small. In practice, we can pretend the derivative, when $z = 0$, is 1 or 0 and the code would work fine.

Rule of thumb for choosing the activation function:

- If the output is zero-one value, when doing binary classification, then the sigmoid activation function is a natural choice for the output layer.
- For other outputs, ReLU (Rectified Linear Unit) is increasingly the default choice of activation function, though sometimes people also use the tanh activation function.

One disadvantage of ReLU is that the derivative is equal to zero when $z < 0$. In practice, this works just fine. However, there is another version of ReLU called the **Leaky ReLU** , which usually works better than the ReLU activation function, but it's not used as much in practice.

The formula is given by

$$a = \max\{0.01 \times z, z\}$$

and it's graphed as follows:

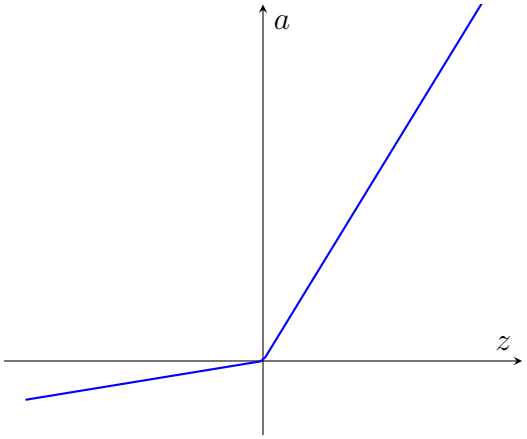

Figure 3.6: Leaky ReLU Function

The advantage for both ReLU and Leaky ReLU is that for a lot of the space of z, the derivative of the activation function is very different from zero. In practice, by using the ReLU activation function, the neural network will often learn much faster than when using the tanh or the sigmoid activation function. The main reason is that there is less of the effect of the derivative of the function going to zero, which slows down learning.

Even though half of z would have derivative equal to z for ReLU, in practice, enough of hidden units will have $z > 0$, so learning can still be quite fast for most training examples.

Although ReLU is widely used and often times the default choice, it is very difficult to know in advance exactly which activation function will work the best for the idiosyncrasies problems. When unsure about which activation function works the best, try them all and evaluate on a holdout validation set or a development set. Then, see which one works better and then go with that activation function.

3.6 Why Non-Linear Activation Functions?

It turns out that for neural network to compute interesting functions, we do need to pick a non-linear activation function.

Suppose given x, we use linear activation function, or in this case, identity activation function, as follows:

- $z^{[1]} = W^{[1]}x + b^{[1]}, a^{[1]} = z^{[1]}$
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, a^{[2]} = z^{[2]}$

If we do this, then the model is just computing $\hat{y}$ as a linear function of the input feature x. We would have

$$\begin{aligned} a^{[1]} &= z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[2]} &= z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ &= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} \\ &= (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]}) \\ &= W'x + b' \end{aligned}$$

In this case, the neural network is just outputting a linear function of the input. For deep neural networks, if we use a linear activation function, or, alternatively, if we do not have an activation function, then no matter how many layers the neural network has, all it's doing is just computing a linear activation function, so we might as well not have any hidden layers.

Suppose we have the following neural network:

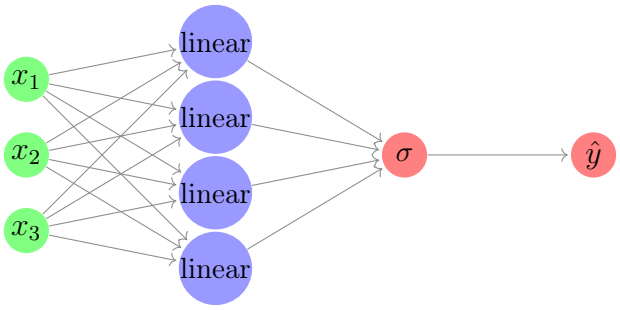

Figure 3.7: Neural Network with Linear Layers

This model is no more expressive than standard logistic regression without any hidden layer. A linear hidden layer is more or less useless because the composition of two linear functions is itself a linear function. Unless we throw a non-linearity, we are not computing more interesting functions even as we go deeper in the network.

There is one player where we might use a linear activation function $g(z) = z$. That's when we are doing machine learning on the regression problem. For example, if we are trying to predict housing prices, where a price $y \in \mathbb{R}$. It might be okay to use a linear activation for the output layer so that $\hat{y}$ is also a real number. However, the hidden units should not use linear activation functions. Other than this, using a linear activation function in the hidden layer, except for some very special circumstances relating to compression, is extremely rare.

3.7 Derivatives of Activation Functions

Sigmoid activation function is given by

$$g(z) = \frac{1}{1+e^{-z}}.$$

We can compute its derivative as follows:

$$\begin{aligned} g'(z) = \frac{\partial}{\partial z} g(z) &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \\ &= \frac{1+e^{-z}-1}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \\ &= \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right) \cdot \frac{1}{1+e^{-z}} \\ &= (1 - \frac{1}{1+e^{-z}}) \cdot \frac{1}{1+e^{-z}} \\ &= g(z) \cdot (1 - g(z)). \end{aligned}$$

Let's sanity check that this expression makes sense. When z is very large. Say, $z = 10$. Then, $g(z) \approx 1$ and

$$\frac{\partial}{\partial z} g(z) \approx 1 \cdot (1-1) = 0.$$

Conversely, when $z = -10, g(z) \approx 0$. From the derivative formula above, we have

$$\frac{\partial}{\partial z} g(z) \approx 0 \cdot (1-0) = 0.$$

When $z = 0, g(z) = 0.5$. In this case,

$$\frac{\partial}{\partial z} g(z) \approx \frac{1}{2} \cdot (1 - \frac{1}{2}) = \frac{1}{4}.$$

In the neural network setting, we use a to denote $g(z)$. Therefore, sometimes we see

$$g'(z) = a(1-a).$$

If we have the value of a, then we can quickly compute the value for $g'(z)$. **tanh activation function** is given by

$$g(z) = \tanh(z).$$

We can compute the derivative as follows:

$$\begin{aligned} \frac{\partial}{\partial} \tanh(z) &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\ &= 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\ &= 1 - \tanh^2(z). \end{aligned}$$

Similar to sigmoid function, we may use a to denote $g(z)$. In this case,

$$g'(z) = 1 - a^2.$$

ReLU activation function is given by

$$g(z) = \max\{0, z\}.$$

The derivative is given by

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$$

Technically, if we define $g'(z) = 1$ when $z = 0$, then $g'(z)$ becomes a sub-gradient of the activation function $g(z)$, which is why gradient descent still works. Therefore, in practice, we use the following derivative:

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Leaky ReLU activation function is given by

$$g(z) = \max\{0.01z, z\}.$$

The derivative, in practice, can be expressed as follows:

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

3.8 Gradient Descent for Neural Networks

In this section, we show how to implement gradient descent for neural network with one hidden layer.

The neural network with a single hidden layer has parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$, where we have $n^{[0]}$ input features, $n^{[1]}$ hidden units, and $n^{[2]} = 1$ output units. Therefore, $W^{[1]}$ has shape $(n^{[1]}, n^{[0]})$, $b^{[1]}$ has shape $(n^{[1]}, 1)$, $W^{[2]}$ has shape $(n^{[2]}, n^{[1]})$, and $b^{[2]}$ has shape $(n^{[2]}, 1)$.

Suppose we are doing binary classification, so the cost function is as follows:

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y),$$

where $\hat{y} = a^{[2]}$.

In this case, our loss function is given by

$$\mathcal{L}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}).$$

The gradient descent algorithm for neural networks could be given as follows:

Algorithm 4 Gradient Descent for Neural Networks

Initialize parameters **randomly** rather than to all zeros.
repeat
Compute predictions $\hat{y}^{(i)}$ for $i = 1, \ldots, m$.
Compute $dW^{[1]} = \frac{\partial J}{\partial db^{[1]}}, dW^{[1]} = \frac{\partial J}{\partial db^{[1]}}, dW^{[2]} = \frac{\partial J}{\partial dW^{[2]}}, db^{[2]} = \frac{\partial J}{\partial db^{[2]}}$.
Update $W^{[1]} = W^{[1]} - \alpha dW^{[1]}$
Update $b^{[1]} = b^{[1]} - \alpha db^{[1]}$
Update $W^{[2]} = W^{[2]} - \alpha dW^{[2]}$
Update $b^{[2]} = b^{[2]} - \alpha db^{[2]}$
until the cost function J converges

In summary, the following are the equations for forward propagation (in vectorized form), assuming we are doing binary classification:

$$
\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) = \sigma(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}X + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})
\end{aligned}
$$

In backward propagation step, the derivatives are given as follows:

$$
\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\
db^{[2]} &= \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} \times g^{[1]'}(Z^{[1]}) \text{ element-wise} \\
dW^{[2]} &= \frac{1}{m} dZ^{[1]} X^T \\
db^{[1]} &= \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

Recall the 2 layer neural network we had:

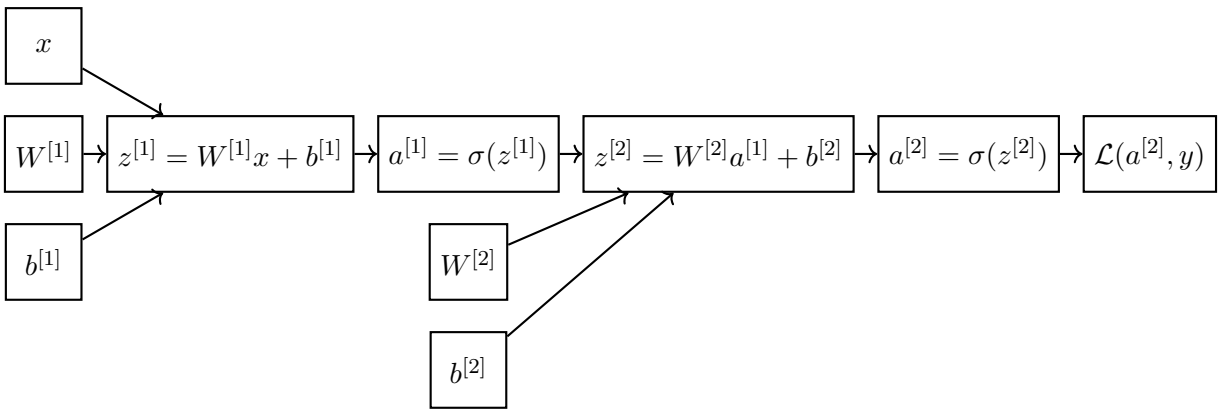

Figure 3.8: 2 Layer Neural Network

Now, we can compute derivatives as follows:

$$
\begin{aligned}
dZ^{[2]} &= a^{[2]} - y \\
dW^{[2]} &= dZ^{[2]} a^{[1]T} \\
db^{[2]} &= dZ^{[2]} \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} \times g^{[1]'}(Z^{[1]}) \text{ element-wise} \\
dW^{[1]} &= dZ^{[1]} x^T \\
db^{[1]} &= dZ^{[1]}
\end{aligned}
$$

The vectorized form can be represented as follows:

$$
\begin{aligned}
Z^{[1]} &= W^{[1]} X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]})
\end{aligned}
$$

where

$$
Z^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix}
$$

The vectorized implementation is as follows:

$$
\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\
db^{[2]} &= \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} \times g^{[1]'}(Z^{[1]}) \text{ element-wise} \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} X^T \\
db^{[1]} &= \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

3.9 Random Initialization

For logistic regression, it was okay to initialize the weights to zero. However, for a neural network, if we initialize the weights to parameters to all zero and apply gradient descent, it will not work.

Suppose we have the following shallow neural network:

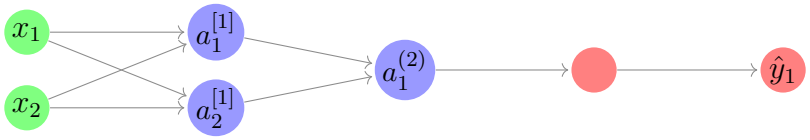

Figure 3.9: Shallow Neural Network Example

In this case, we have two input features and so $n^{[0]} = 2$. Also, we have two hidden units in the first hidden layer, so $n^{[1]} = 2$. Thus, $W^{[1]} \in \mathbb{R}^{2\times 2}$ and $b^{[1]} \in \mathbb{R}^{2\times 1}$. Initializing $b^{[1]}$ to all zeros is okay, but not for $W^{[1]}$.

Suppose we have

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, W^{[2]} = \begin{bmatrix} 0 & 0 \end{bmatrix},$$

then $a_1^{[1]} = a_2^{[1]}$ and $dZ_1^{[1]} = dZ_2^{[1]}$. If two neurons in the same layer are equal, then they are going to have the same rate of change and update from the gradient. We can proof by induction to show that after t iterations, two hidden units would be computing exactly the same function, which means there is no point to have more than one hidden unit. With the same changes after each iteration, the two neurons will act as if they are the same, thus the additional neuron does not add more information. We call this error the **symmetry breaking problem**.

The solution to this is to initialize parameters randomly as follows:

$$W^{[1]} = np.random.randn((2, 2)) * 0.01.$$

We multiply the random generated number by 0.01 to make sure that the weights are initialized to very small random values. If we are using hyperbolic tangent or sigmoid function, then if the weights are too large, $Z^{[1]} = W^{[1]}X + b^{[1]}$ would be very large and thus we are more likely to end up at flat parts of the activation function, where the gradient is very small. In this case, gradient descent will be very slow, thus the learning. On the other hand, if we do not have sigmoid or tanh activation functions throughout the neural network, then this is less of an issue.

Sometimes, there can be better constants than 0.01. When training a shallow neural network, say it has only one hidden layer, set constant to 0.01 would probably work okay. But when we are training on a very deep neural network, then we might to choose a different constant, which will be discussed in later sections.

It turns out that our bias term $b^{[1]}$ does not have the symmetry breaking problem, so it is okay to initialize $b^{[1]}$ to just zeros as follows:

$$b^{[1]} = np.zeros((2, 1)).$$

We can initialize $W^{[2]}, b^{[2]}$ in a similar way.

Note that so long as $W^{[1]}$ is initialized randomly, we start off with different hidden units computing different functions and thus we would not have the symmetry breaking problem.

Chapter 4

Deep Neural Networks

4.1 Deep L-layer Neural Network

We say that logistic regression is a "shallow" model, where is does not have any hidden layers.

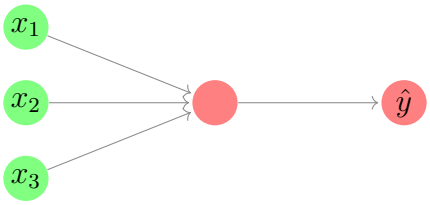

Figure 4.1: Logistic Regression

On the other hand, a neural network, shown below, with 5 hidden layers is a much deeper model.

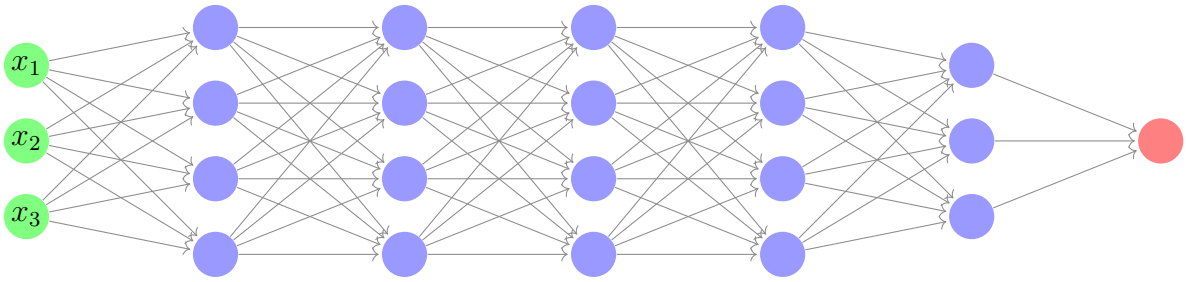

Figure 4.2: Neural Network with 5 Hidden Layers

A neural network with 1 hidden layer is called a "2 layer Neural Network."

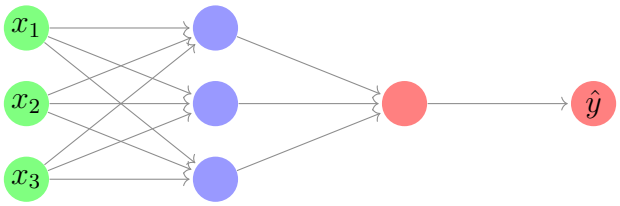

Figure 4.3: Neural Network with 1 Hidden Layers

To clarify the notations about deep neural network, let's use the following 4 layer neural network as an example:

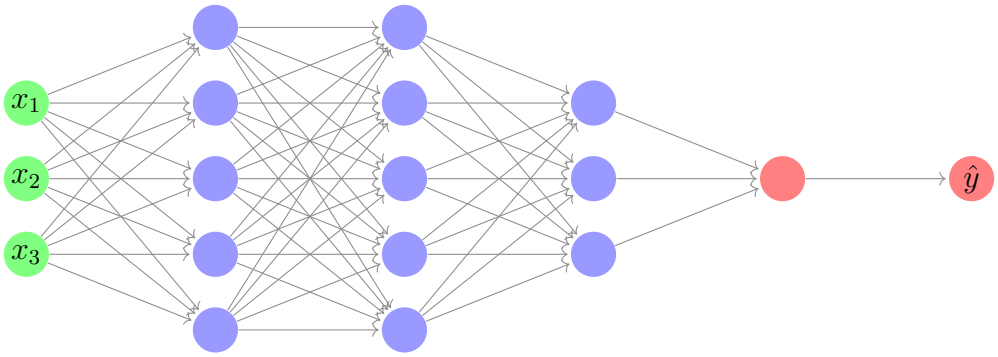

Figure 4.4: 4 Layer Neural Network

There are 3 hidden layers. The number of units in these hidden layers are $5, 5, 3$ and there is 1 output unit.

- We use capital L to denote the number of layers. In this case, $L = 4$.
- We use $n^{[l]}$ to denote the number of nodes/units in layer l. For example, $n^{[0]} = n_x = 3, n^{[1]} = n^{[2]} = 5, n^{[3]} = 3, n^{[4]} = n^{[L]} = 1$.
- For each layer l, we use $a^{[l]}$ to denote the activations in layer l. For example, $a^{[l]} = g^{[l]}(z^{[l]})$. In any case, $x = a^{[0]}$ and $\hat{y} = a^{[L]}$.
- We use $w^{[l]}$ to denote the weights for $z^{[l]}$.
- We use $b^{[l]}$ to denote the biases for $z^{[l]}$.

4.2 Forward Propagation in a Deep Network

Given a training example x, we can compute the activations of the first layer as follows:

$$
\begin{aligned}
z^{[1]} &= w^{[1]}x + b^{[1]} = w^{[1]}a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]})
\end{aligned}
$$

$$
\begin{aligned}
z^{[2]} &= w^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]})
\end{aligned}
$$

$$
\begin{aligned}
z^{[3]} &= w^{[3]}a^{[2]} + b^{[3]} \\
a^{[3]} &= g^{[3]}(z^{[3]})
\end{aligned}
$$

$$
\begin{aligned}
z^{[4]} &= w^{[4]}a^{[3]} + b^{[4]} \\
a^{[4]} &= g^{[4]}(z^{[4]}) = \hat{y}
\end{aligned}
$$

The general rule is that $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$ and $a^{[l]} = g^{[l]}(z^{[l]})$.
The vectorized form would be the following:

$$Z^{[1]} = W^{[1]}X + b^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$

$$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$$
$$A^{[3]} = g^{[3]}(Z^{[3]})$$

$$\hat{Y} = g^{[4]}(Z^{[4]}) = A^{[4]}$$

In this implementation of vectorization, there is a for-loop where we compute values of $l = 1, \ldots, 4$. In this case, there is no way to implement without an explicit for-loop. Therefore, it's okay to have a for-loop for forward propagation.

4.3 Getting Matrix Dimensions Right

One way to do this is to pull a piece of paper and work through the dimensions of matrices we are working with.

Suppose we have the following neural network:

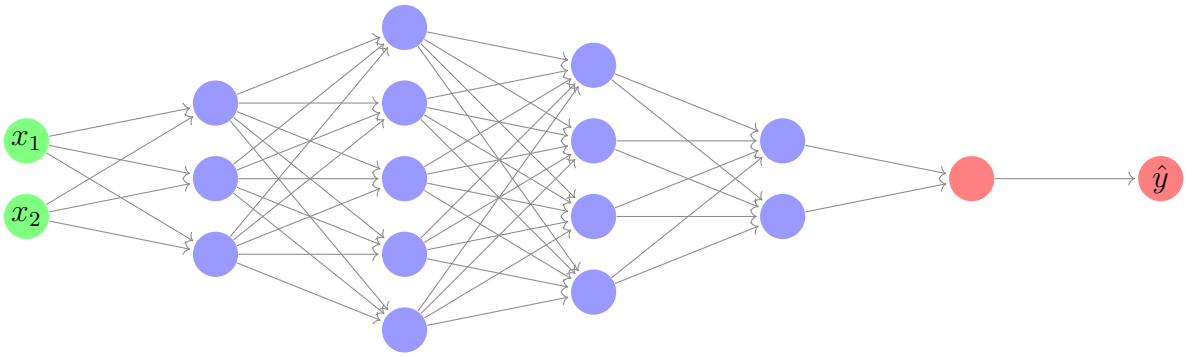

Figure 4.5: 5-Layer Neural Network

For this neural network, we have

$$n^{[0]} = 2, n^{[1]} = 3, n^{[2]} = 5, n^{[3]} = 4, n^{[4]} = 2, n^{[5]} = 1.$$

When implementing forward propagation, we have

$$z^{[1]} = w^{[1]}x + b^{[1]},$$

which is the activations for the first hidden layer. Therefore, the dimensions of $z^{[1]}$ is $(n^{[1]}, 1)$, or $(3, 1)$. x has dimensions $(2, 1)$, or more generally $(n^{[0]}, 1)$. By the

rule of matrix multiplication, we can infer that $W^{[1]}$ has dimensions $(3, 2)$, or more generally, $(n^{[1]}, n^{[0]})$. $b^{[1]}$ has dimension $(3, 1)$, or more generally, $(n^{[1]}, 1)$.

Similarly, $w^{[2]}$ has dimensions $(5, 3)$. $z^{[2]}$ has dimensions $(5, 1)$. $a^{[1]}$ has dimensions $(3, 1)$. $w^{[3]}$ has dimensions $(4, 5)$. $w^{[4]}$ has dimensions $(2, 4)$. $w^{[5]}$ has dimensions $(1, 2)$.

Since $a^{[l]}$ is the obtained from element-wise operation of $g^{[l]}$ on $z^{[l]}$, $a^{[l]}$ and $z^{[l]}$ should have the same dimensions.

When implementing backward propagation, $dw^{[l]}$ should have the same dimensions as $w^{[l]}$ and $db^{[l]}$ should have the same dimensions as $b^{[l]}$.

The general formula to check dimensions is given as follows:

$$\begin{aligned} w^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ b^{[l]} &: (n^{[l]}, 1) \\ dw^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ db^{[l]} &: (n^{[l]}, 1) \end{aligned}$$

For vectorized implementation, the dimensions are a little different. For the forward propagation, we have

$$Z^{[1]} = W^{[1]}X + b^{[1]}.$$

In this case, $Z^{[1]}$ has dimensions $(n^{[1]}, m)$. The dimensions of $w^{[1]}$ stays the same, which are $(n^{[1]}, n^{[0]})$. Now, X has dimensions $n^{[0]}, m)$. $b^{[1]}$ is still $(n^{[1]}, 1)$, but broadcasting gives us $(n^{[1]}, m)$. We can summarize dimensions as follows:

$$Z^{[l]}, A^{[l]}, dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$$

4.4 Why Deep Representation?

What is a deep neural network computing? Suppose we are building a system for face recognition, or face detection. Then, the first layer of neural network could be an edge detector for the input picture. It can then group edges together to form parts of a face in the next hidden layer. Finally, by putting together different parts of a face like an eye, an ear, or a nose, the next layer, it can then try to recognize or detect different types of faces.

This type of simple to complex hierarchical representation, or compositional representation, applies to other types of data than images and face recognition. For example, if we are trying to build a speech recognition system, it's hard to visualize speech. However, if we input an audio clip, then the first layer of a neural network might learn to detect low level audio wave form features, such as if the tone is going up. Then, by composing low level wave forms, the neural network might learn to detect basic units of sounds, which is called phonemes in Linguistics. By composing these together, it learns words. Then, neural network learns to recognize sentences or phrases.

Whereas early layers of a deep neural network compute relatively simple functions of the input, such as where the edge is, by the time we get deep into the network, we can do surprisingly complex things.

Circuit theory pertains thinking about what types of functions we can compute with different AND gates, OR gates, NOT gates, and other logic gates. Informally, there are functions we can compute with a "small" (small number of hidden units) L-layer deep neural network that shallower networks require exponentially more hidden units to compute. For example, if we want to compute $y = x_1 XOR x_2 XOR \ldots, XOR x_n$. It would take $\mathcal{O}(\log(n))$ layers to build the XOR tree, each layer with small number of hidden units. However, if we restrict the number of layers to 1, then we would need $\mathcal{O}(2^n)$ hidden units, which is exponentially large.

4.5 Building Blocks of Deep Neural Networks

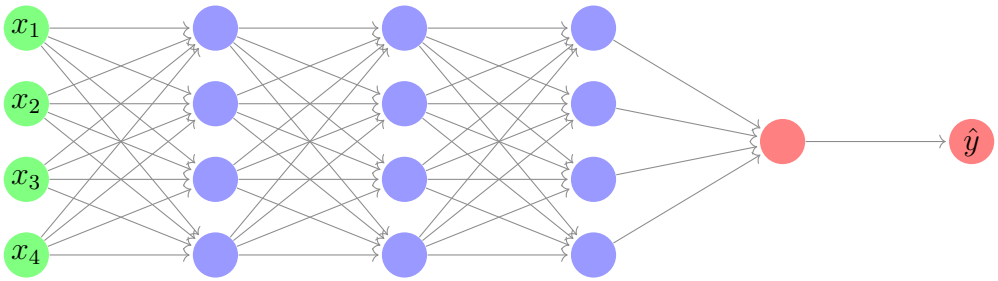

Figure 4.6: 4-Layer Neural Network Example

Suppose we have the above neural network. Let's look into the computations focusing on just the layer l. For layer l, we have $w^{[l]}, b^{[l]}$.

For the forward propagation, it has input $a^{[l-1]}$ and outputs $a^{[l]}$ as follows:

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

Meanwhile, we cache the result $z^{[l]}$, and potentially $w^{[l]}, b^{[l]}$, as it would be useful for backward propagation step later.

Then, for the backward propagation, we have input $da^{[l]}, cache(z^{[l]})$ and want output $da^{[l-1]}, dw^{[l]}, db^{[l]}$.

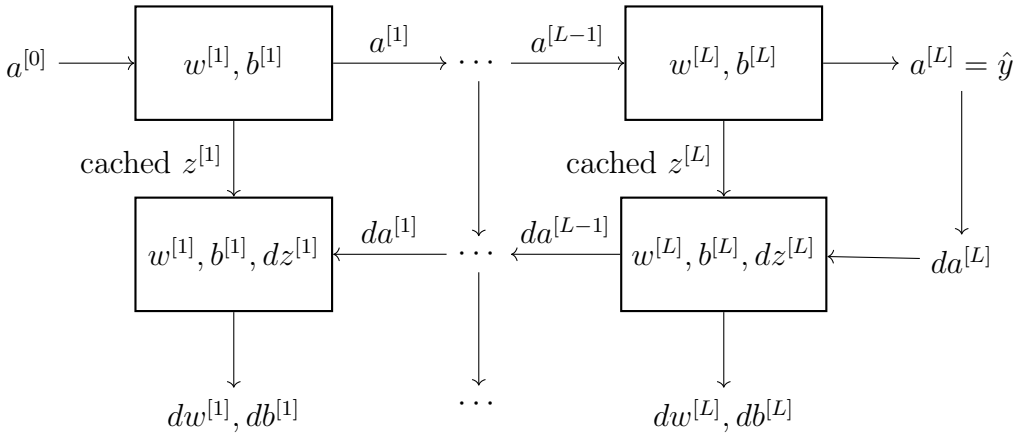

Figure 4.7: One Full Iteration of Gradient Descent for Neural Network

4.6 Forward and Backward Propagation

In this section, we discuss the implementation of forward and backward propagation.

The vectorized implementation of forward propagation is given below:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

where we initialize $A^{[0]} = X$.

For backward propagation, given input $da^{[l]}$, we output

$$da^{[l-1]}, dW^{[l]}, db^{[l]}.$$

The implementation is given as follows:

$$\begin{aligned} dz^{[l]} &= da^{[l]} \times g^{[l]'}(z^{[l]}) \text{ element-wise} \\ dW^{[l]} &= dz^{[l]}a^{[l-1]T} \\ db^{[l]} &= dz^{[l]} \\ da^{[l-1]} &= W^{[l]T}dz^{[l]} \end{aligned}$$

If we take the definition of da and plug into the definition of dz, we get

$$dz^{[l]} = W^{[l+1]T}dz^{[l+1]} \times g^{[l]'}(z^{[l]}),$$

which is consistent with previous presentation.

The vectorized version is given below:

$$\begin{aligned} dZ^{[l]} &= dA^{[l]} \times g^{[l]'}(Z^{[l]}) \text{ element-wise} \\ dW^{[l]} &= \frac{1}{m}dZ^{[l]}A^{[l-1]T} \\ db^{[l]} &= \frac{1}{m}np.sum(dZ^{[l]}, axis = 1, keepdims = True) \\ dA^{[l-1]} &= W^{[l]T}dZ^{[l]} \end{aligned}$$

In the context of binary classification, we can derive that

$$da^{[l]} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} = \frac{-y}{a} + \frac{(1-y)}{(1-a)},$$

which is used for initialization of backward recursion.

In the vectorized implementation, we initialize as follows:

$$dA^{[l]} = \left(\frac{-y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{(1-a^{(1)})}\right) + \cdots + \left(\frac{-y^{(m)}}{a^{(m)}} + \frac{(1-y^{(m)})}{(1-a^{(m)})}\right)$$

4.7 Parameters vs Hyperparameters

Parameters in the learning algorithm are weights $W^{[l]}$'s and biases $b^{[l]}$'s. Hyperparameters determine the real parameters and they are

- learning rate α
- number of iterations
- number of hidden layers L
- number of hidden units $n^{[1]}, n^{[2]}, \ldots$
- Choice of activation function

In later sections, we will see our hyperparameters in deep learning, such as momentum term, the mini-batch size, various forms of regularization parameters, and so on.

Applied deep learning is a very empirical process where we might have an idea about the learning rate $\alpha = 0.01$. Then, we implement it and see how it works. Then, we adjust the learning rate accordingly. We need to try out many things and see which work.

Also, as we make progress on a problem, it is quite possible that the best values for hyperparameters might change – this might be caused by the computer infrastructure. The rule of thumb is that for every few months when we are working on a problem for an extended period of time, just try a few values of hyperparameters and check if there are better values for them.

4.8 What does this have to do with the brain?

People often make analogy between deep learning and the human brain. When doing neural network, we implement forward and backward propagation. A single biological neurons in the brain receives electric signals from other neurons, does a simple thresholding computation, and then if this neuron fires, it sends a pulse of electricity down the axon to other neurons. Therefore, there is a loose analogy between a single neuron in a neural network and biological neuron.

However, even today, the neuroscientist have almost no idea what even a single neuron is doing. A single neuron appears to be more complex than we are able to characterize with neuroscience. It is completely unclear today whether the human brain uses an algorithm does anything like backward propagation or gradient descent, or there are some fundamentally different learning principle that the human brain uses.

Part II

Improving Deep Neural Network: Hyperparameter Tuning, Regularization, and Optimization

Chapter 5

Practical Aspects of Deep Learning

5.1 Setting up Machine Learning Application

Making good choices of setting up training, development, and test sets can make a huge difference in helping us quickly find a good high-performance neural network. When starting off on a new problem, it is almost impossible to correctly guess the right values for hyperparameter choices. In practice, applied machine learning is a highly iterative process We often start with an idea. Then we code it up and run an experiment to get back a result that tells how well one particular network works. Based off the outcome, we then refine ideas and change the choices. We keep iterating this process and find a better and better neural network.

5.1.1 Train/Dev/Test Sets

Setting up dataset well, in terms of train, development, and test sets, can make us more efficient at going around the iterative process.

For a given training data, traditionally, we will carve off some portion of it to be the training set, some to be hold-out cross validation, or development, set, and some final portion to be the test set. We keep on training our algorithm on training set and use hold-out cross validation set to see which model performs the best on the “dev" set. After having done this long enough, when we have a final model that we want to evaluate, we can take the best model and evaluate it on the test set in order to get an unbiased estimate of how well our algorithm is doing.

In the previous era of machine learning, it was a common practice to take all data and split it according to **70%/30%** train-test split if we do not have explicit “dev" set. If we have “dev" set, then people do a **60%/20%/20%** split. Several years ago, this was widely considered the best practice in machine learning.

However, in the modern big data era, where we might have a million examples in total, the trend is that our “dev" and test sets become a much smaller percentage of the total. The reason is that the goal of the dev set, or development set, is that we are testing different algorithms on it and see which ones works better, so the dev set just has to be big enough for us to evaluate different algorithm choices and quickly decide which ones are better. For example, if we have a million training examples, we might decide to have 10K examples in the dev set, which is more than enough. In a similar vein, the main goal of the test set is, given the final algorithm, to give us a pretty confident estimate of how well it is doing. Again if we have a

million training examples, we might decide to have 10K examples to evaluate the algorithm. In this case, we are splitting with **98%/1%/1%** ratios. If we have more than a million examples, then we might end up having **99.5%/0.25%/0.25%**, or **99.5%/0.4%/0.1%** split.

Furthermore, more and more people are training on **mismatched train and test** distributions. Suppose we are building an app that lets users upload a lot of pictures and the goal is to find pictures of cats in order to show the users. The training set could come from cat pictures downloaded off the Internet, but our dev and test sets might comprise of cat pictures from users using the app. It turns out that a lot of webpages have very high resolution, very professional, nicely framed pictures of cats. But the users may be uploading blurrier, lower resolution images taken with a cell phone camera in a more casual condition. Therefore, the distributions of data may be different. The rule of thumb in this case is to make sure the **dev and test sets come from the same distribution**.

Finally, it might be okay to not have a test set if we don't need an unbiased estimate to evaluate our final model. In this case, we go over the iterative process by training on the training set, trying different model architectures, and evaluating models on the dev set.

5.2 Bias/Variance

Bias and Variance is one of those concepts that's easily learned but difficult to master. In the Deep Learning era, there has been less discussion of Bias/Variance trade-off.

Suppose we have a dataset with two labels of 2D data points. If we fit a line to the data with high bias, then we say this is underfitting the data. On the other hand, if we fit a very complex classifier, then we have a high variance, in which case we overfit the data. With the medium level of model complexity, then we are fitting "just right."

With high dimensional data, we cannot visualize the decision boundary. Instead, we will look at different metrics to try to understand bias and variance. For the cat classification problem, we will look at **train set error** and **dev set error**.

Let's say the train set error is 1% and dev set error is 11%. In this case, we are doing pretty well on the training set, but relatively poorly on the development set. We might have overfitted the training set, so the model does not generalize well to the hold-out cross validation set. This has **high variance**.

Suppose, instead, the train set error is 15% and dev set error is 16%. Assume humans can achieve roughly 0% error, then it looks like the algorithm is not even doing very well on the training set. In this case, we are underfitting the data and thus it has **high bias**.

Now, suppose the train set error is 15% and dev set error is 30%. Then the algorithm has **high bias and high variance**. This could happen because we choose a very simple model, yet overfit a few training examples. On the other hand, suppose train set error is 0.5% and dev set error is 1%. This model has **low bias and low variance**. These categorizations all rely on the assumption that the optimal error, sometimes called **Bayes Error**, is nearly 0%. We would have had a different analysis if we have a much higher Bayes Error.

The trade-off is summarized in the figure below:

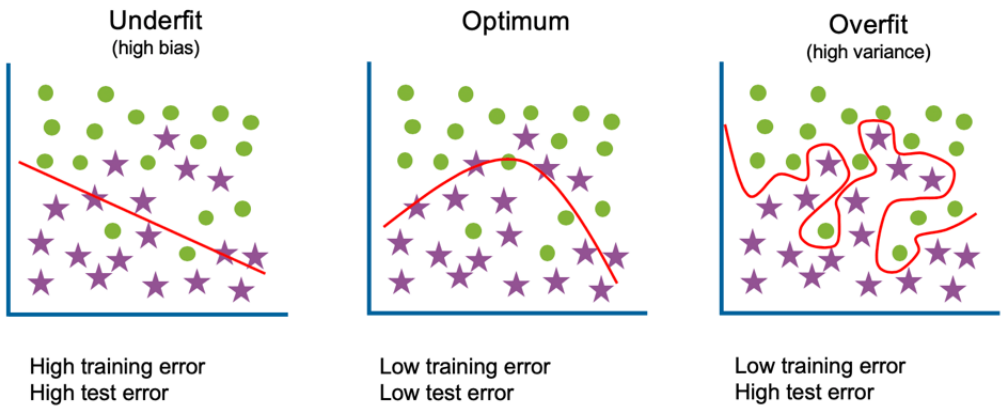

Figure 5.1: Bias/Variance Trade-off

5.3 Basic Recipe for Machine Learning

In this section, we present the ways to improve algorithm more systematically, depending on whether it has high bias or high variance issues.

After training the algorithm, we check the following:

1. **If the algorithm have high bias** by looking at the training set performance. If it does have high bias – not fitting training set well – we can try

   - bigger network, such as more hidden layers or more hidden units.
   - train algorithm longer.
   - try some more advanced optimization algorithms.
   - try a different neural network architecture that is better suited for the problem.

2. Once we have reduced bias to an acceptable value, we ask **if we have variance problem** by looking at dev set performance – are we able to generalize from a pretty good training set performance to having a pretty good dev set performance? If we have a high variance problem, then we can

   - get more data.
   - regularization.
   - try a more appropriate neural network architecture.

3. Then we go back to check **if we have high bias again** and hopefully we find something with both low bias and low variance.

In pre deep learning era, we reducing bias would often lead to an increased variance, or vice versa. However, with deep learning technique, we can almost always reducing one without hurting the other. It almost never hurts to try a deeper neural network, as long as we regularize well.

5.4 Regularizing Neural Network

5.4.1 Regularization

If we have a high variance problem, then one of the first things to try is regularization. The other way to get rid of high variance is to get more training data, which is also quite reliable. However, we cannot always get more training data, as it could be expensive. Adding regularization will often help prevent overfitting, or reduce variance in the network.

Let's develop the idea using logistic regression. Recall that for logistic regression, the cost function is given by

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}),$$

where $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$.

With regularization term, we use the following cost function instead:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2,$$

where $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$. This is called **L2 regularization**, which is the most common type of regularization. In practice, we could also add a regularization term for b, but it is usually omitted. w is usually pretty high dimensional parameter vector, so most likely high variance means we are not fitting these parameters well, whereas b is just a single number. Almost all the parameters are in w, rather than in b. Therefore, adding a regularization for b will not make much difference.

We also have **L1 regularization** given by

$$\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w| = \frac{\lambda}{2m}\|w\|_1.$$

If we use L1 regularization, then w will end up being sparse, meaning the w vector will have a lot of zeros in it. Some people say this can help compressing the model – as a set of parameters are zero, we need less memory to store the model, although, in practice, this helps only a little bit.

In either case, λ is called the **regularization parameter** and usually we set this using the development set by trying a variety of values and see which one does the best.

For the neural network, the cost function is given by

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2,$$

where $\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(w_{i,j}^{[l]})^2$. This matrix norm is called the **Frobenius norm**, or $L2$ norm, of the matrix. Previously, to implement gradient descent, we would compute $dw^{[l]}$ by backprop and we update $w^{[l]}$ with $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$. Now, with the regularization term, we add

$$\frac{\lambda}{m}w^{[l]}$$

to the gradient.

In this case, we multiply $w^{[l]}$ by $1 - \frac{\alpha\lambda}{m}$ and then subtract learning rate times the original gradient to do the update. Since we multiply $w^{[l]}$ by a number smaller than 1 every time we update the parameter, L2 norm is also called “weight decay."

5.4.2 Why Regularization Reduces Overfitting?

If we set λ to be really big, then we will incentivize the model to set the weight matrix to be reasonably close to zero, meaning $w^{[l]} \approx 0$. This zeroes out impacts of many hidden units. Then, the simplified neural network becomes a much smaller neural network. It gets closer and closer to as if we are just using logistic regression. This moves high variance problem towards high bias issue, but hopefully there will be an intermediate value of λ that results in the “just right" case.

Here is another way to gain intuition. Suppose we use tanh activation function, meaning $g(z) = \tanh(z)$. If λ is really large, then the set of parameters in $w^{[l]}$ will be quite small, resulting in relatively small $z^{[l]}$ because $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$. This puts $g(z)$ to the linear regime of the tanh function., where $g(z)$ is roughly linear. Therefore, every layer would be roughly linear, as if it is just linear regression. We showed before that if every hidden layers are linear, then the whole network is just a linear network. As a result, the deep neural network cannot fit very complicated non-linear decision boundaries that overfit the dataset.

5.4.3 Dropout Regularization

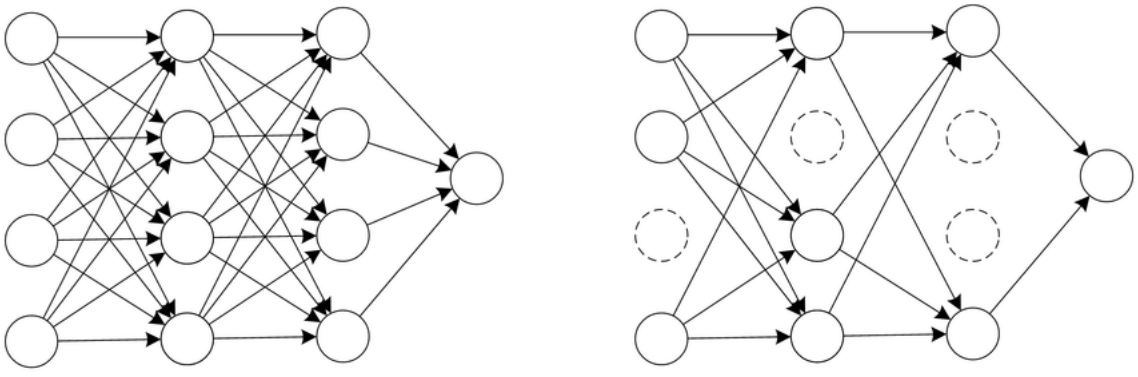

Figure 5.2: Neural Network with Dropout

With dropout, we go through each layer of the network and set some probability of eliminating a node in the neural network. Suppose for each layer, we have 50% chance of keeping each node and 50% chance of removing each node. After eliminating nodes, we remove all the outgoing edges from that node, so we end up with a much diminished network. Then, we do backprop, training one example on this much diminished network. On different examples, we toss a set of coins again and keep a different set of nodes and then dropout different set of nodes. For each training example, we train it using one of the reduced networks. As we train individual

examples on much smaller networks, this gives a reason why we end up being able to regularize the network.

Now, we discuss a implementation of dropout, called **inverted dropout technique**. We illustrate with a single layer, layer $l = 3$, where we use $d3$ to represent the dropout vector for layer 3 and keep_prob to represent the probability of keeping a node.

$$d3 = np.random.rand(a3.shape[0], a3.shape[1]) < kepp\_prob$$
$$a3 = np.multiply(a3, d3)$$
$$a = a/keep\_prob$$

Suppose $keep\_prob = 0.8$ and we have 50 hidden units in the third hidden layer. Now, we have 80% chance of keeping the node and 20% chance of removing the node for each individual node. On average, we have 10 units shut off, so $a^{[3]}$ is expected to be reduced by 20%. In order to maintain the expected value of $z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$, we need to divide $a^{[3]}$ by 0.8. This ensures that no matter what we set keep_prob to, the expected value of $a^{[3]}$ remains the same, and thus $z^{[4]}$. It turns out that at test time, the inverted dropout technique makes test time easier because we have less of a scaling problem.

At test time, we are given some $a^{[0]} = x$ and we want to make a prediction. We do not use dropout at test time, as we do not want the output to be random. If we implement dropout at test time, this just adds noise to the predictions.

5.4.4 Understanding Dropout

Many of the first successful implementations of dropouts were in computer vision, where input sizes are really big. We almost always don't have enough data, so we are almost always overfitting. Dropout randomly knocks out units in the network, so it's as if on every iteration, we are working with a smaller neural network. Using smaller neural network seems like it should have a regularizing effect. In this section, we provide another intuition.

With dropout, some input units are eliminated, so a hidden unit cannot rely on any one feature because any one feature could go away at random. Consequently, the unit will be more motivated to spread out weights and give a little bit of weight to each of its inputs. By spreading out the weights, this will tend to have an effect of shrinking the squared norm of the weights. Similar to L2 regularization, it helps prevent overfitting. It actually turns out that dropout can formally be shown to be an adaptive form of L2 regularization, but L2 penalty on different weights are different depending on the size of the activation being multiplied.

It turns out we can use different values of keep_prob for different layers. To reduce overfitting of the matrix with largest parameter size, we can have a low keep_prob for that layer with large dimension of parameters. We can also drop out some of the inputs, though, in practice, we usually don't do that, so $keep\_prob = 1.0$ is quite common for the input layer.

Actually, unless the algorithm is overfitting, we would not bother to use dropout. One big downside of dropout is that the cost function J is no longer well-defined. Therefore, it's hard to plot the value of cost function against the number of iterations, so we lose a debugging tool to find bugs. In this case, we can turn off dropout by setting keep_prob to 1 and run the code to make sure that the cost value is

monotonically decreasing. Then, we turn on the dropout and hope that we did not introduce bugs in the code.

5.4.5 Other Regularization Methods

Suppose we are fitting a cat classifier. If we are overfitting, more training data can help, but getting more training data can be expensive. We can augment our data by, for example, flipping it horizontally and adding it to the training set. However, since the training set is now a bit redundant, this isn't as good as if we had collected an additional set of new independent examples. Also, we can random crops, distortions, or translations of an image. This is called **data augmentation** and this is an inexpensive way to give our algorithm more data and reduce overfitting. For optical character recognition, we can also augment dataset by imposing random rotations and distortions to a given digit.

One other technique that is often used is **early stopping**. As we run gradient descent, we plot training error, or cost J, against the number of iterations, which should decrease monotonically. We also plot dev set error. We usually find that dev set error decreases for a while and then it will increase. In this case, we will stop training on the neural network when dev set error starts to increase. Why does this work? When we just start the gradient descent iteration, the parameters w will be close to zero. As we iterate, w will get bigger than bigger. By stopping halfway, we only have a mid-side $\|w\|_F^2$. Similar to L2 regularization, by picking a neural network with smaller norm for parameters w, hopefully the neural network is overfitting less.

Machine learning process is comprised of several different steps.

- We want the algorithm to find a set of parameters w, b to optimize cost function J. We use tools like gradient descent, Adam optimizer, etc. to do it.

- Not overfit by using regularization, getting more data, etc..

In machine learning, we already have many hyperparameters to tune. Therefore, it is easier to think about when we have one set of tools for optimizing cost function J, where all we care about is finding w, b such that $J(w, b)$ is as small as possible. Then, reduce overfit is another completely separate task. This principle is called **orthogonalization**. The idea is that we want to think about only one task at a time. This introduces the downside of early stopping as it couples two tasks together. We can no longer work on two problems independently. By stopping early, we are breaking the optimization of cost function J and, simultaneously, trying to not overfit. In this case, rather than using early stopping, we can just use L2 regularization to train as long as possible. The downside of this, though, is that we need to try a lot of values of hyperparameters, which makes it more computationally expensive.

5.5 Setting Up Optimization Problem

5.5.1 Normalizing Inputs

By normalizing inputs, we can speed up the training for neural network. We can do so by subtracting the mean as follows:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
$$x = x - \mu$$

The second step is to normalize the variances as follows:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} * *2$$
$$x = x/\sigma$$

If we use this to scale our training data, we use the same μ, σ to normalize the test set.

Why do we bother normalizing inputs? Recall that the cost function is defined as

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

It turns out if we use unnormalized input features, it is more likely that the cost function would look like a squished out bar. If the features are in very different scales, then parameters will take very different values, causing cost function to be elongated. Therefore, we would need a small learning rate for the cost function to oscillate back and forth until it reaches the minimum. If we normalize the features, then the cost function will be more symmetric. In this case, no matter where we start, gradient descent can take much larger steps and go straight to the minimum. The comparison could be summarized by the following figure:

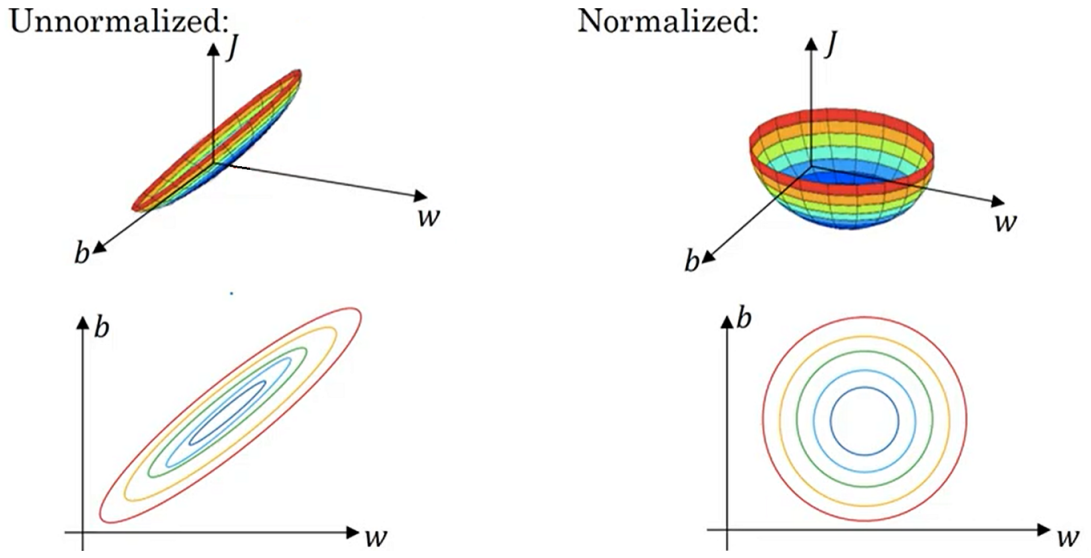

Figure 5.3: Normalizing Inputs

5.5.2 Vanishing/Exploding Gradients

One big problem of training neural networks, especially deep neural networks, is data vanishing and exploding gradients. The derivatives can sometimes get either very big or very small, which makes training difficult.

Suppose we are training a very deep neural network with parameters $w^{[1]}, \ldots, w^{[l]}$, $b^{[l]} = 0$, and activation function $g(z) = z$ for the sake of simplicity. Now, the output would be

$$\hat{y} = \left(\prod_{l=1}^{L} w^{[l]} \right) x.$$

Suppose each feature matrix $w^{[l]}$ is

$$w^{[l]} = \begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix}.$$

Then, $\hat{y}$ can be rewritten as

$$\hat{y} = w^{[L]} \begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix}^{L-1} x.$$

Effectively, $\hat{y} = \gamma^{L-1} x$. If $\gamma > 1$ and L is very large for deep network, then $\hat{y}$ will be very large. In fact, this grows exponentially. Conversely, if $\gamma < 1$, then activation values will decrease exponentially as a function of the number of layers L of the network. A similar argument could be used to show that gradients will also increase or decrease exponentially as a function of the number of layers.

With some of the modern neural networks, $L = 150$. Microsoft got great results with 152 layer neural network. With such a deep neural network, if the activations or gradients increase or decrease exponentially as a function of L, then the values could be really big or really small, which makes training difficult.

5.5.3 Weight Initialization for Deep Networks

A partial solution to the vanishing/exploding gradients would be a better or more careful choice of random initialization for the neural network. Suppose we have the following single neuron network:

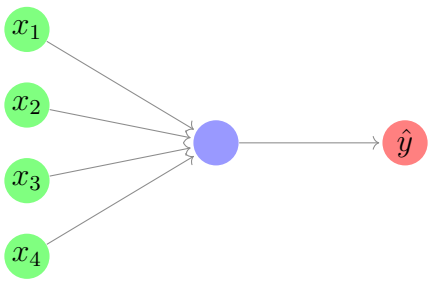

Figure 5.4: Single Neuron Example

In this case, by setting $b = 0$, we have

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n.$$

The larger the n is, the smaller we want w_i to be. One reasonable thing to do is to set the variance of w to be

$$Var(w_l) = \frac{1}{n^{[l-1]}},$$

where n is the number of input features that is going into the neuron.

In practice, we can do the following:

$$w^{[l]} = np.random.randn(shape) * np.sqrt(1/n^{[l-1]}).$$

It turns out if the activation function is ReLU, then setting variance of

$$Var(w_l) = \frac{2}{n^{[l-1]}}$$

works a little bit better.

By doing this, the values $w^{[l]}$ are not too much bigger or smaller than 1, so it does not explode or vanish too quickly. For other variants, if we use tanh activation function, then we use

$$Var(w_l) = \frac{1}{n^{[l-1]}}.$$

This is called **Xavier initialization**. Another version is given by

$$Var(w_l) = \frac{2}{n^{[l-1]} + n^{[l]}}.$$

If we want, this variance could also be a hyperparameter that we try to tune, but this is less important relative to other hyperparameters.

5.5.4 Numetical Approximation of Gradients

When implementing backprop, there is a technique call **gradient checking** that helps make sure that the implementation is correct. To build up to gradient checking, we will first discuss how to numerically approximate computations of gradients.

Recall that the formall definition of derivative is given by

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}.$$

For $f(x) = x^3$ and $x = \theta$. We nudge θ to both left and right to $\theta - \epsilon$ and $\theta + \epsilon$ for some small number $\epsilon > 0$. Therefore,

$$g(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

gives a much better approximation to the derivative at θ than using $\theta + \epsilon$ alone. We can show that the approximation error is on the order of $\mathcal{O}(\epsilon^2)$, where the Big-O constant happens to be 1. When $\theta = 1, \epsilon = 0.01$, our approximation is

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001,$$

and the real derivative is $g(\theta) = 3\theta^2 = 3$. The approximation error is thus 0.0001, whereas using $\theta + \epsilon$ alone gives us 0.301 approximation error. If we use one-sided

approximation, then the approximation error is on the order of $\mathcal{O}(\epsilon)$. Since ϵ is a small number, the one-sided approximation is less accurate than the two-sided approximation.

However, it turns out that in practice, the two-sided approximation runs twice as slow as the one-sided approximation, but it's worth it to use two-sided method, as it's more accurate.

5.5.5 Gradient Checking

Gradient Checking (Grad Check) is a technique that helps find bugs in the implementations of backprop and verify that the implementation is correct.

Suppose our neural network has parameters $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$. The first thing we need to do for gradient checking is to take all these parameters and reshape into a giant vector θ. Now the cost function becomes a function of θ:

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta) = J(\theta_1, \theta_2, \ldots).$$

Similarly, we take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape them into a giant vector $d\theta$, with the same dimension as θ. Is $d\theta$ the gradient of J?

Algorithm 5 Gradient Checking

for each i **do**

Compute

$$d\theta_{approx}[i] = \frac{J(\theta_1, \ldots, \theta_i + \epsilon, \ldots) - J(\theta_1, \ldots, \theta_i - \epsilon, \ldots)}{2\epsilon}.$$

Compute $d\theta[i] = \frac{\partial J}{\partial \theta_i}$.

end for

Then, we check if $d\theta_{approx}$ and $d\theta$ are approximately equal. We could check Euclidean distance and then normalize it as follows:

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}.$$

In practice, we use $\epsilon = 10^{-7}$. If the formula gives a value of 10^{-7} or smaller, then the derivative approximation is likely to be correct. If it's 10^{-5}, maybe it's okay, but we need to take a careful look. If the formula gives value of 10^{-3}, then most likely there is a bug. We should look at the vector to find if there is a specific value of i, which $d\theta_{approx}[i]$ is very different from $d\theta[i]$.

5.5.6 Gradient Checking Implementation Notes

In this section, we cover some practical tips about how to implement grad check correctly to debug our code. The tips are as follows.

- Firstly, we do not use grad check in training. It's only used to debug. Computing $d\theta_{approx}[i]$ is a very slow computation.

- Secondly, if an algorithm fails grad check, we look at individual components to try to identify the bug. For example, if $db^{[l]}$'s are pretty far off, but $dW^{[l]}$'s are quite close, then the bug might be the way we compute db, and vice versa.
- We need to remember the regularization term when computing the gradient.
- Grad check does not work with dropout. There isn't an easy-to-compute cost function J that dropout is doing gradient descent on. Therefore, we need to implement grad check without dropout, or we can fix the pattern of nodes dropped and use grad check to check grads for that pattern are correct, though in practice we do not usually do this.
- Maybe our implementation of backprop is only correct when w, b are close to 0 and it gets more inaccurate when w, b become larger. Though not done often in practice, we can run grad check at random initialization and then run grad check again after some number of iterations.

Chapter 6

Optimization Algorithms

In this chapter, we present optimization algorithms that would enable us to train neural networks much faster. Deep learning tends to work the best in the regime of big data, when we train our neural network on a huge dataset. However, this is just slow. Having a good optimization algorithm can spped up the efficiency of training.

6.1 Mini-batch Gradient Descent

Previously, we have shown that vectorization allows us to efficiently compute on m training examples without an explicit for-loop. However, if m is very large – say, 5 million – then the process can still be slow.

With the implementation of gradient descent, we need to process the entire training set before we take one little step of gradient descent. We can split training sets into little baby training sets, which are called **mini-batches**, each having 1000 training examples. We can denote the set of training examples $x^{(1)}, \ldots, x^{(1000)}$ as $X^{\{1\}}$ and all the way until $X^{\{5000\}}$. Similarly, we split Y accordingly into $Y^{\{1\}}, \ldots, Y^{\{5000\}}$. Therefore, mini-batch t is comprised of $X^{\{t\}}, Y^{\{t\}}$, where $X^{\{t\}}$ has dimensions $(n_x, 1000)$ and $Y^{\{t\}}$ has dimensions $(1, 1000)$.

Batch gradient descent processes the entire batch of training examples all at the same time, whereas mini-batch gradient descent processes a single mini-batch $X^{\{t\}}, Y^{\{t\}}$ at the same time.

Algorithm 6 Mini-batch Gradient Descent

for each i **do**

Perform forwardprop on $X^{\{t\}}$ for $l = 1, \ldots, L$:

$$Z^{[l]} = W^{[l]} X^{\{t\}} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]}$$

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)} + \frac{\lambda}{2 \cdot 1000} \sum_l \|w^{[l]}\|_F^2$.

Backprop to compute gradients w.r.t $J^{\{t\}}$.

Update weights with $W^{[l]} := W^{[l]} - \alpha dW^{[l]}, b^{[l]} := b^{[l]} - \alpha db^{[l]}$

end for

When we have a large training set, mini-batch gradient descent runs much faster than batch gradient descent. Pretty much every one in deep learning uses it.

6.2 Understanding Mini-batch Gradient Descent

In this section, we show more details of how to implement mini-batch gradient descent and gain a better understanding of what it's doing and why it works.

With batch gradient descent, we expect the cost to go down as we train for more iterations. On mini-batch gradient descent, if cost might not decrease on every additional iteration, but it shows a decreasing trend overtime. The reason why mini-batch cost values are more noisy is because $X^{\{t\}}, y^{\{t\}}$ might be a easy mini-batch while $X^{\{t+1\}}, y^{\{t+1\}}$ is a harder mini-batch. If there are mislabeled examples inside, then the cost will be higher. The comparison is shown in the figure below:

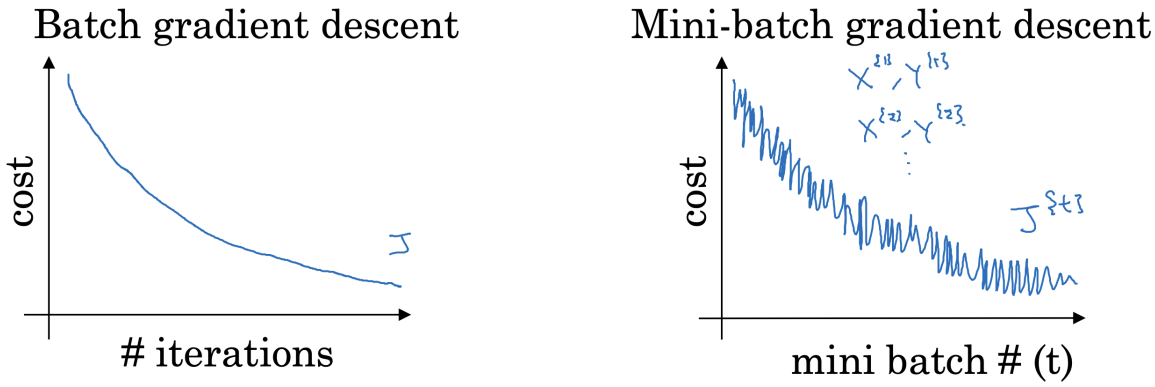

Figure 6.1: Batch vs Mini-batch Gradient Descent

As we use mini-batch gradient descent, we would need to choose mini-batch size. If mini-batch size is m, then we are doing batch gradient descent where

$$(X^{\{t\}}, y^{\{t\}} = (X, Y).$$

On the other hand, if we choose mini-batch size to be 1, then we are doing **stochatic gradient descent** where every training example is its own mini-batch, where

$$(X^{\{t\}}, y^{\{t\}} = (x^{(1)}, y^{(1)}), \ldots, (x^{(n_x)}, y^{(n_x)})$$

In practice, we use mini-batch size in between 1 and m. If we use batch gradient descent, where mini-batch size is m, then we are processing huge training set at every iteration, so it takes a long time to run each iterations. On the other hand, if we use stochastic gradient descent, even though the noise can be reduced by using smaller learning rate, we lose all the speed-ups from vectorization. By choosing a size in between 1 and m, we have the fastest learning, where we do get a lot of vectorization and make progress without processing the entire training set.

The convergence is shown below, where purple represents stochastic gradient descent, blue represents batch gradient descent, and green represents mini-batch size in between 1 and m:

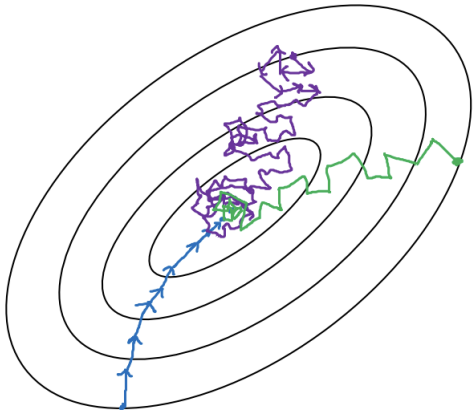

Figure 6.2: Convergence Comparison among Different Batch Sizes

Here is the general guidelines of choosing the mini-batch size.

- If we have small training set, then just use batch gradient descent. By small, it means $m \leq 2000$.
- Typical mini-batch size would be from 64 to 512. Because of the way computer memory is laid out and accessed, sometimes the code runs faster if the mini-batch size is a power of 2. Therefore, we choose mini-batch size from $\{64, 128, 256, 512\}$.
- Make sure mini-batch $X^{\{t\}}, y^{\{t\}}$ fit into CPU/GPU memory.

At the end, the mini-batch size is also a hyperparameter we can tune. We can run gradient descent with mini-batch size of some power of 2 and see which one does the best job.

6.3 Exponentially Weighted Averages

In order to understand optimization algorithms that run faster than gradient descent, we need to use **exponentially weighted averages**, or exponentially weighted moving averages in Statistics.

For example, temperature data could be pretty noisy. If we want to compute the trends, or the moving average of the temperature, we can do the following:

$$\begin{aligned} v_0 &= 0 \\ v_1 &= 0.9v_0 + 0.1\theta_1 \\ v_2 &= 0.9v_1 + 0.1\theta_2 \\ v_3 &= 0.9v_2 + 0.1\theta_3 \\ &\vdots \\ v_t &= 0.9v_{t-1} + 0.1\theta_t \end{aligned}$$

where θ_i represents the temperature on day i. This is the exponentially weighted average of the daily temperature. More generally, we can represent the moving average as

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t.$$

6.4 Understanding Exponentially Weighted Averages

Suppose $t = 100$ and $\beta = 0.9$. By expanding iterative equations $v_{100} = 0.9v_{99} + 0.1\theta_{100}$, we get the following:

$$\begin{aligned} v_{100} &= 0.1 + 0.9v_{99} \\ &= 0.1 + 0.9(0.1 + 0.9v_{98}) \\ &= 0.1 + 0.9(0.1 + 0.9(0.1 + 0.9v_{97})) \\ &= 0.1\theta_{100} + 0.1(0.9)\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + 0.1(0.9)^4\theta_{96} + \cdots \end{aligned}$$

All the coefficients add up to 1 or close to 1, up to a detail called **bias correction**. In this example,

$$0.9^{10} \approx 0.35 \approx \frac{1}{e}.$$

One fact we know is that

$$(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}.$$

Therefore, it takes 10 days for the weight to decay to less than about a third of the weight of the current day. If $\beta = 0.98$, then

$$0.98^{50} \approx \frac{1}{e},$$

as $\epsilon = 0.02$.

In general, we are averaging over

$$\frac{1}{1-\beta}$$

days, where $\epsilon = 1 - \beta$.

We can implement this as follows:

$v_\theta = 0$
repeat{
    get next θ_t
    $v_\theta := \beta v_\theta + (1-\beta)\theta_t$
    }

This takes very little memory since we keep on overwriting one variable with the formula. Also, this just takes one line of code, which is very efficient, especially when we try to compute averages for multiple variables.

6.5 Bias Correction in Exponentially Weighted Averages

Bias correction can make our computation of exponentially weighted averages more accurate.

Suppose $\beta = 0.98$. Then, from the formula we showed,

$$v_1 = 0.98v_0 + 0.02\theta_1 = 0.02\theta_1.$$

Similarly,

$$\begin{aligned} v_2 &= 0.98v_1 + 0.02\theta_2 \\ &= 0.98(0.02)\theta_1 + 0.02\theta_2 \\ &= 0.0196\theta_1 + 0.02\theta_2. \end{aligned}$$

Neither v_1 nor v_2 would be a good estimate of the temperature. The way to modify the estimate to make it much more accurate, especially during the initial phase of the estimate, we can instead take

$$\frac{v_t}{1-\beta^t}.$$

For example, suppose $t = 2$, then

$$1 - \beta^t = 1 - (0.98)^2 = 0.0396.$$

Therefore, our estimate of the temperature on day 2 becomes

$$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

and this becomes a weighted average of θ_1 and θ_2 and removes the bias.

Also, as t becomes large, β^t approaches 0 and thus the bias correction makes almost no difference. When t is small, when we are warming up the estimations, the bias correction could help us make a better estimate of the temperature.

6.6 Gradient Descent with Momentum

Gradient descent with momentum almost always works faster than the standard gradient descent algorithm. The basic idea is that we compute an exponentially weighted average of gradients, and then use that gradient to update weights instead.

Suppose the contour plot of the cost function is given as follows:

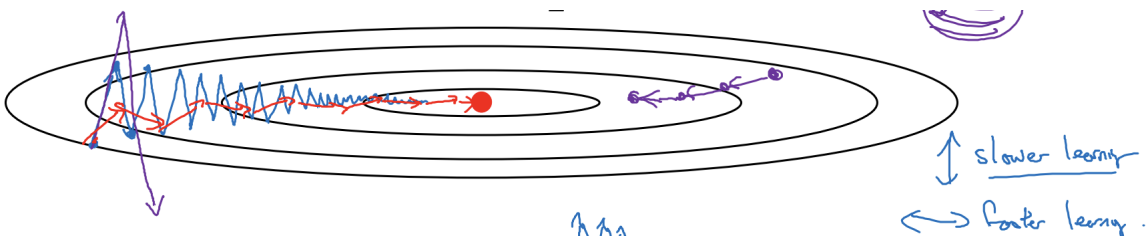

Figure 6.3: Gradient Descent with Momentum Contour Plot

For the standard gradient descent, shown in blue line, gradient descent takes a lot of steps and slowly oscillates towards the minimum. This up and down oscillations slow down gradient descent and prevent us from using a much larger learning rate. If we use a much larger learning rate, then we could overshoot like the purple line in 6.3.

On the vertical axis, we want the learning to be a bit slower because we do not want to overshoot oscillations. However, on the horizontal axis, we want faster

learning because we want to aggressively move from left to right towards the red dot.

Gradient descent with momentum smooths out the steps of gradient descent. For gradients, the oscillations on the vertical direction tend to average out to close to zero. This will average out positive and negative numbers and so the average will be close to zero. On the horizontal direction, all the directions are pointing to the right, so the average in the horizontal direction will still be pretty big. Therefore, gradient descent with momentum will take steps with much smaller oscillations in the vertical direction and more directed to moving quickly in the horizontal direction, as shown by the red line in 6.3. The algorithm is given below:

Algorithm 7 Gradient with Momentum

$v_{dW} = 0, v_{db} = 0$.
for each iteration t **do**
Compute dW, db on the current mini-batch (omitted superscripts).
Compute $v_{dW} := \beta v_{dW} + (1 - \beta)dw$.
Compute $v_{db} := \beta v_{db} + (1 - \beta)db$.
Update parameters as

$$W := W - \alpha v_{dW}$$
$$b := b - \alpha v_{db}$$

end for

Imagine we are optimizing a bowl-shaped function. Then, the gradient terms are providing acceleration to a ball rolling down the hill. The momentum terms represent the velocity. Therefore, we can accelerate down the bowl and gain momentum.

For this algorithm, we have hyperparameters α, learning rate, and β, which controls the exponentially weighted average. In practice, $\beta = 0.9$, which averages last 10 gradients, works very well. It appears to be a pretty robust value. Also, in practice, people do not do bias correction because after just 10 iterations, the moving average will have warmed up and is no longer a bias estimate.

In some literature, we see that v_{dW}, v_{db} are represented as

$$v_{dW} := \beta v_{dW} + dw$$
$$v_{db} := \beta v_{db} + db$$

where $(1 - \beta)$ is omitted. The net effect is that v_{dW} and v_{db} end up being scaled by a factor of $1 - \beta$. Therefore, when we perform the gradient descent updates, α has to be changed by a corresponding value of

$$\frac{1}{1 - \beta}.$$

this requires us to tune the learning rate α differently.

6.7 RMSprop

RMSprop (root mean squared prop) can similarly speed up gradient descent. In this section, we show how it works.

Algorithm 8 RMSprop

$v_{dW} = 0, v_{db} = 0$.
for each iteration t **do**
Compute dW, db on the current mini-batch (omitted superscripts).
Compute $S_{dW} := \beta_2 v_{dW} + (1 - \beta_2) dw^2$ (element-wise).
Compute $S_{db} := \beta_2 v_{db} + (1 - \beta_2) db^2$ (element-wise).
Update parameters as

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \epsilon}$$
$$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$$

end for

The algorithm is presented as follows:

Small $\epsilon > 0$ is added to ensure numerical stability in case S_{dW} or S_{db} equals to zero. $\epsilon = 10^{-8}$ is a good default. With these terms, we hope S_{dW} will be relatively small and S_{db} will be relatively large, so we can slow down the updates on the vertical direction and speed up the updates on the horizontal direction. As db will be relatively large and dW will be relatively small, dividing db by $\sqrt{S_{db}}$ will damp out the oscillations. The net effect can be shown in the figure below, with the green line:

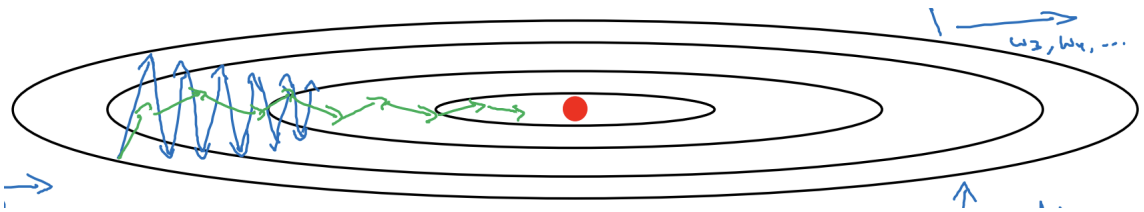

Figure 6.4: RMSprop Convergence

In this case, we can use larger α without diverging in the vertical direction. In practice, the vertical/horizontal direction could be sets of parameters in high-dimensional parameter vector.

This idea was not first proposed in a academic paper, but in a Coursera course by Geoffrey E. Hinton.

6.8 Adam Optimization Algorithm

RMSprop and Adam optimizers have shown to work well on a wide range of deep learning architectures. Adam optimizer basically takes momentum and RMSprop and puts them together.

The algorithm is presented as follows:

Algorithm 9 Adam Optimization Algorithm

$v_{dW} := 0, S_{dW} := 0, v_{db} := 0, S_{db} := 0.$
for each iteration t **do**
Compute dW, db on the current mini-batch (omitted superscripts).
Compute $v_{dW} := \beta_1 v_{dW} + (1-\beta_1)dW, v_{db} := \beta_1 v_{db} + (1-\beta_1)db$
Compute $S_{dW} := \beta_2 v_{dW} + (1-\beta_2)dw^2; S_{db} := \beta_2 v_{db} + (1-\beta_2)db^2$ (element-wise).
Bias correction on v as $v_{dW}^{corrected} := v_{dW}/(1-\beta_1^t); v_{db}^{corrected} := v_{db}/(1-\beta_1^t)$
Bias correction on S as $S_{dW}^{corrected} := S_{dW}/(1-\beta_1^t); S_{db}^{corrected} := S_{db}/(1-\beta_1^t)$
Update parameters as

$$W := W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$$
$$b := b - \alpha \frac{v_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

end for

This is a common used algorithm that's proven to be very effective for many neural network architectures. In this algorithm, there are many hyperparameters.

- Learning rate α. This ususally needs to be tuned.
- Moving average of dW, β_1, has a default value 0.9.
- β_2 has a recommended value of 0.999 by the author of Adam algorithm paper.
- The choice of ϵ does not matter much, but the author of the Adam paper recommends 10^{-8}.

Adam stands for **Adaptive moment estimation**, where β_1 is computing the mean of the derivatives – the first moment – and β_2 is computing exponentially weighted average of the squares – the second moment.

6.9 Learning Rate Decay

If we slowly decrease the learning rate α, then we can wander around a tight region close to the optimal point. In the initial steps of learning, we can afford to take much bigger steps, but as learning approaches convergence, then having a smaller learning rate allows us to take smaller steps.

Recall that one epoch is one pass through the dataset. We can set learning rate to

$$\alpha = \frac{1}{1 + \text{decayRate} \times \text{epochNumber}} \alpha_0,$$

where α_0 is some initial learning rate. Here, the decayRate becomes another hyperparameter we might need to tune.

Suppose $\alpha_0 = 0.2$ and $decayRate = 1$. We can construct the following example to better illustrate the idea:

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04

There are other learning rate decay methods.

- Exponentially decay
$$\alpha = \beta^{\text{epochNumber}}\alpha_0$$
where $0 \leq \beta < 1$.
- $$\alpha = \frac{k}{\sqrt{\text{epochNumber}}}\alpha_0 \quad or \quad \alpha = \frac{k}{\sqrt{t}}\alpha_0$$
where k is some constant and t is the mini-batch number.
- Discrete staircase decrease of learning rate α where we decay after some number of steps.
- Manual decay. If we are training one model and the model is taking hours or days to train, we can tune α hour-by-hour or day-by-day. This only works when we are training a small number of models.

6.10 The Problem of Local Optima

In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima. It turns out if we train a neural network, most points of zero gradients are not local optima, but saddle points. In 20000 dimensional space, for a point to be a local minimum, we would need to have gradient concaving down in all directions, which has a very small chance. It's more likely the case where one direction is curving down and another is curving up, giving us a saddle point. It is very unlikely to get stuck in a bad local optima, so long as we train on a reasonably large neural network. This tells us a lot of intuitions we have for low-dimensional spaces really don't transfer to high-dimensional spaces, where our learning algorithms are operating over.

It turns out **plateaus** can really slow down learning. A plateau is a region where the derivatives is close to zero for a long time. As the plateau is nearly flat, it takes a very long time to slowly find our way off the plateau, as shown below:

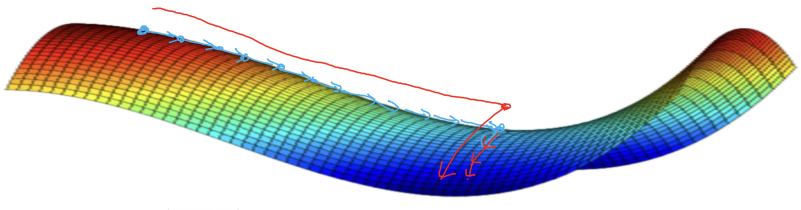

Figure 6.5: Plateau Example

Chapter 7

Hyperparameter Tuning and More

7.1 Hyperparameter Tuning

One of the painful things about training a deep neural network is the sheer number of hyperparameters we have to deal with. Out of all sorts of hyperparameters, the learning rate α is the most important one to tune. Then, we would prioritize momentum term β, though its default is 0.9, mini-batch size, and number of hidden units. The third importance would be learning rate decay. When using Adam optimizer, we almost never tune $\beta_1, \beta_2, \epsilon$, which are, by default, $0.9, 0.999, 10^{-8}$. But we can tune these if we wish.

To tune the hyperparameters, how do we set values to explore? In earlier generations of machine learning algorithms, if we have two hyperparameters, it was a common practice to sample the points in a grid and systematically explore these values like the following figure:

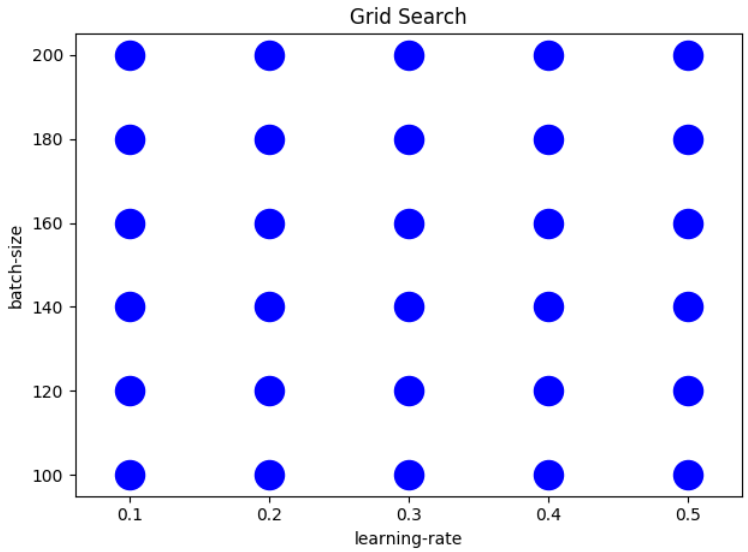

Figure 7.1: Grid Search

This works okay if the number of hyperparameters is relatively small. In deep learning, we tend to choose points at random and try out hyperparameters on the randomly chosen set of points. It's difficult to know in advance which hyperparameters are going to be the most important for the problem we work on. If we do

grid search on one important hyperparameter and one unimportant hyperparameter, then we are trying out less values of the important hyperparameter. If we were to sample at random, then we will have tried out more distinct values of the more important hyperparameter and find out which value works better. The random search could be visualized as follows:

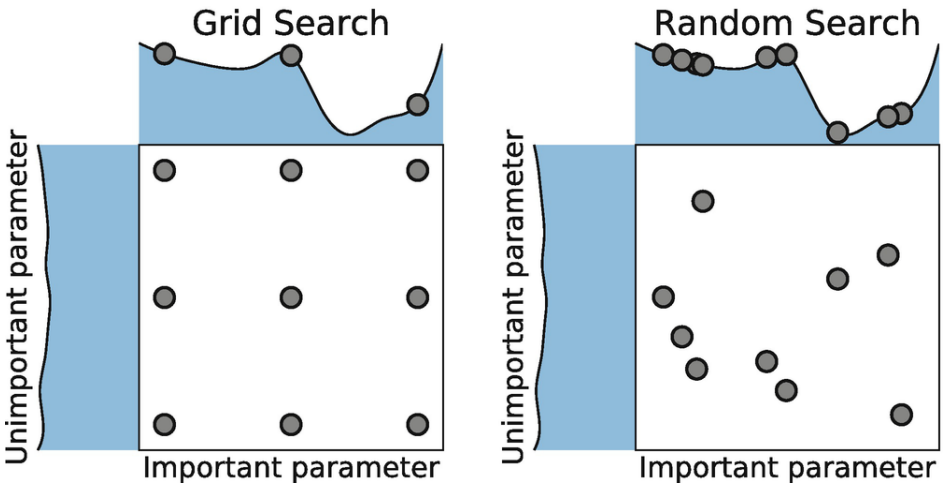

Figure 7.2: Random Search

Another common practice for sampling hyperparameters is to use a **course to fine** sampling scheme. We zoom in into a smaller region and sample more densely within the space at random for the region that generally does a good job. We can visualize as follows:

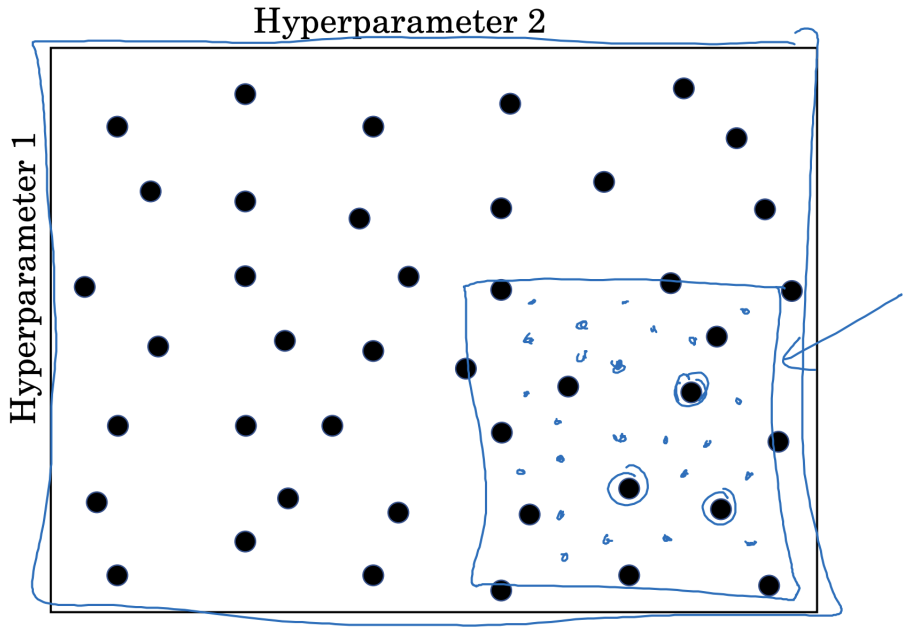

Figure 7.3: Course to Fine Search

By trying out these hyperparameters, we can pick values that allow us to do the best on the training set objective, or do the best on the development set.

7.2 Using an Appropriate Scale to Pick Hyperparameter

Sampling at random does not mean sampling uniformly at random over the range of valid values. It is important to pick the appropriate scales to explore hyperparameters.

Suppose we try to choose the number of hidden units and we believe a good range of values are $n^{[l]} = 50, \ldots, 100$. Then we can randomly pick values within this range. If, instead, we try to decide on the number of layers in the neural network L and we think it should be somewhere between 2 to 4. Then, sampling uniformly within $2, 3, 4$ might be reasonable, or even using a grid search, where we explicitly evaluate the values $2, 3, 4$. These two examples are reasonable things to do. However, this is not true for all hyperparameters.

Suppose we are searching for the hyperparameter α and we suspect that 0.0001 might be on the low end and it could be as high as 1. If we draw a number line from 0.0001 to 1 and sample values uniformly at random over the number line, then about 90% of the values would be between 0.1 and 1. We only use 10% of resources to search for values between 0.0001 and 0.1. Instead, it seems more reasonable to search for hyperparameters on log scale, where we have $0, 0001, 0.001, 0.01, 0.1, 1$. We then search uniformly at random on the log scale, which we dedicate more resources to search between 0.0001 and 0.1. The way to implement this is as follows:

$$r = -4 * np.random.rand()$$
$$\alpha = 10^r$$

where $r \in [-4, 0]$ and $\alpha \in [10^{-4}, 0]$. More generally, if the end points of possible values are 10^a and 10^b, we can sample r uniformly at random between a and b.

One other tricky thing is sampling hyperparameter β for exponentially weighted averages. Suppose we suspect $0.9 \leq \beta \leq 0.999$. Recall that using $\beta = 0.9$ is like averaging over the last 10 values, whereas using $\beta = 0.999$ is like averaging over last 1000 values. It also does not make sense to search samples on the linear scale. A good way to thing about this is that we want to explore $0.001 \leq 1 - \beta \leq 0.1$. Therefore, we can use the example we figured out before to do the search, where we sample $r \in [-3, -1]$ and we set $1 - r = 10^r \Rightarrow \beta = 1 - 10^r$. When β is close to 1, the sensitivity of the results we get changes. If β goes from 0.9 to 0.90005, this is hardly any change in our results, as we are still averaging roughly 10 values. However, if β goes from 0.999 to 0.9995, this will have a huge impact on what the algorithm is doing, as we are averaging from 1000 examples to 2000 examples. This causes us to sample more densely in the regime where β is close to 1, or $1 - \beta$ is close to 0.

7.3 Hyperparameters Tuning in Practice

There are two major schools of thoughts where people go about hyperparameter searching. One way is we **babysit one model**. We usually do this if we have a huge dataset but not a lot of computational resources, i.e. CPU, GPU., so we can only afford to train only one model or a very small number of models at a time. For example, at day 0, we might initialize parameters at random and start training. At the end of day 1, if we see the algorithm is learning quite well, we can probably

increase the learning rate and see how it does. After day 2, if it still does well, then we can fill the momentum term a bit or decrease the learning rate. Every day, we look at the algorithm performance and nudge up and down hyperparameters.

The other approach would be if we **train many models in parallel**. We have a set of hyperparameters and let the algorithm run for days while keeping track of a metric. We plot the curve of metric at the end. A second model would generate another curve for the metric. At the end, we can quickly pick one set of hyperparameter values that works the best.

Chapter 8

Batch Normalization

This technique can make our neural network much more robust to the choice of hyperparameters. It does not work for all neural networks, but when it does, it can make the hyperparameter search much easier and also make training go much faster. This also enable us to much more easily train even very deep networks.

8.1 Normalizing Activations in a Network

Batch Normalization was created by two researchers, Sergey Loffe and Christian Szegedy.

We saw before that data normalization could turn the contour of learning problem from very elongated shape to more round, which enables algorithms like gradient descent to optimize easier. This works by normalizing input features. How about a deeper network? If we want to train on parameters $w^{[3]}, b^{[3]}$, it would be nice if we can normalize $a^{[2]}$. Technically, batch normalization, or batch norm in short, would normalize $z^{[2]}$. There are debates about whether we should normalize before the activation function, $z^{[2]}$, or after applying the activation function, $a^{[2]}$. In practice, normalizing on $a^{[2]}$ is done much more often.

Given some intermediate values in our neural network, $z^{[l](1)}, \dots, z^{[l](m)}$. We will compute the following for layer l:

$$\begin{aligned}
\mu &= \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)} \\
\sigma &= \frac{1}{m}\sum_{i=1}^{m}(z^{[l](i)} - \mu)^2 \\
z^{[l](i)}_{norm} &= \frac{z^{[l](i)} - \mu}{\sqrt{\sigma + \epsilon}} \\
\tilde{z}^{[l](i)} &= \gamma z^{[l](i)}_{norm} + \beta
\end{aligned}$$

where $\epsilon > 0$ is added for numerical stability and γ, β are learnable parameters of the model. γ, β allow us to set the mean and variance of $\tilde{z}$ to the values we want. In fact, if $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$, then γ, β will invert the normalization equation and $\tilde{z}^{[l](i)} = z^{[l](i)}$. After normalization, we will use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$ for later computations of the neural network.

8.2 Fitting Batch Norm into a Neural Network

Suppose we have the following neural network:

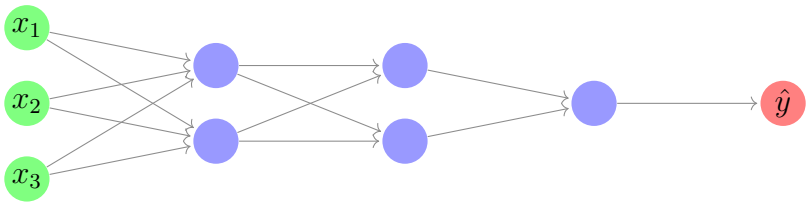

Figure 8.1: Batch Norm Example

With batch norm, we would compute the first two layers as follows:

$$x \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm(BN)}]{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \xrightarrow[\text{BatchNorm(BN)}]{\beta^{[2]},\gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]}.$$

In either case, we use normalized value $\tilde{z}$. The parameters are $w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}$ and $\beta^{[1]}, \gamma^{[1]}, \ldots, \beta^{[L]}, \gamma^{[L]}$. These β's have nothing to do with the momentum term. We might compute $d\beta^{[l]}$ and update $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$. We can also use Adam or RMSprop to update the parameters β, γ, not just gradient descent. In deep learning frameworks, we usually do not need to implement batch norm by ourselves.

In practice, batch norm is applied along with mini-batches of training set. We would do batch normalization on one batch at a time as follows:

$$X^{\{1\}} \xrightarrow{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm(BN)}]{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \rightarrow \ldots$$

Previously, we showed that the parameters for each layer l are $w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$, all of which have dimensions $(n^{[l]}, 1)$. Notice that $z^{[l]}$ is computed as

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}.$$

During the batch norm step, we compute the means of $z^{[l]}$'s and subtract out the mean. Thus, adding any constant to all examples in the mini-batch does not change anything, as the constant will be cancelled out by the mean subtraction step. Therefore, when applying batch norm, we can eliminate $b^{[l]}$'s. We end up using $\beta^{[l]}$ to decide the mean of $\tilde{z}^{[l]}$.

Algorithm 10 Gradient Descent for Batch Normalization

for t=1,...,numMiniBatches **do**

Compute forward prop on $X^{\{t\}}$, where we use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$.

Use backprop to compute $dw^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$ (as $b^{[l]}$ goes away).

Update parameters as

$$w^{[l]} := w^{[l]} - \alpha dw^{[l]}; \beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}; \gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$$

end for

This also works with gradient descent with momentum, RMSprop, or Adam.

8.3 Why does Batch Norm Work?

Batch normalization makes weights deeper in the neural network more robust to changes to weights in earlier layers of the neural network.

In case of **covariate shift**, we would not expect our learning algorithm to do well on finding the decision boundary for binary classification task. The idea is that if we learn some x to y mapping, then if the distribution of x changes, we might need to retrain the learning algorithm. This is true even if the mapping function remains unchanged. How does this relate to neural network? Suppose we have the following deep neural network:

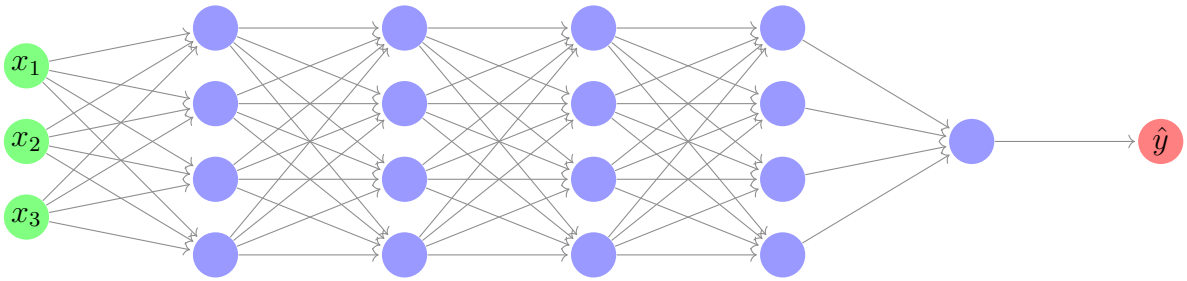

Figure 8.2: Deep Neural Network Example

Consider the learning process of the third hidden layer. The network has learned parameters $w^{[3]}, b^{[3]}$. From the perspective of the third hidden layer, it gets some values $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$. The third hidden layer takes in these values and find a way to map them into $\hat{y}$. As $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ change, the values of $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$ will also change. Therefore, the third hidden layers suffers from the covariate shift. Batch norm reduces shifting of distribution of previous hidden units. The values of $z^{[1]}, z^{[2]}, z^{[3]}, z^{[4]}$ will change, but batch norm ensures that the mean and variance of these values remain the same. It limits the amount to which updating parameters in the earlier layers can affect the distribution of values that the third hidden layer takes, thus making input values more stable.

It turns out batch norm also has some regularization effect.

- Each mini-batch $X^{\{t\}}$ has values $z^{[l]}$ scaled by the mean and variance computed on just that mini-batch, as opposed to computed on the entire dataset. In this case, the mean and variance have a bit noise in them.
- This adds some noise to the values $z^{[l]}$ to $\tilde{z}^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations. This forces the downstream hidden units not to rely too much on any one previous hidden units.
- This has a slight regularization effect – as the noise is quite small.
- By using larger minibatch size, we are reducing the noise and therefore also reducing the regularization effect.

We do not turn batch norm as a regularization. We use it as a way to normalize hidden unit activations and therefore speed up learning. The regularization is an almost unintended side effect.

8.4 Batch Norm at Test Time

Batch norm handles data for one mini-batch at a time. At test time, when we try to evaluate the network, we might not have a mini-batch of examples. We might be processing one single example at a time. Therefore, we need to do something slightly different to ensure our predictions make sense.

In order to apply neural network at test time, we need separate estimates of μ, σ^2, which is usually done using exponentially weighted average across mini-batches. Suppose we go through mini-batches $X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$, where we get

$$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}, \ldots.$$

The exponentially weighted average of $\mu^{\{i\}[l]}$'s becomes μ we use. Similarly, by computing the exponentially weighted average of

$$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]}, \ldots,$$

we get σ^2. Then, at test time, we compute $z^{(i)}_{norm}$ as follows:

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

using the μ, σ obtained above. Then, we compute $\tilde{z}^{(i)}$ as normal, using the β, γ parameters we have learned during the neural network training process. In practice, this process is pretty robust to the exact way we used to estimate μ, σ^2. Any way of reasonably estimating μ, σ^2 would work fine at test time.

Chapter 9

Multi-class Classification

9.1 Softmax Regression

Softmax regression is the generalization of logistic regression, where we can make predictions on one of multiple classes, rather than just two classes.

Suppose we want to recognize cats (1), dogs (2), baby chicks (3), and others (0) as shown below:

Figure 9.1: Multi-class Classification Example

In this case, there are $C = 4$ classes and the indices are $(0, 1, 2, 3)$. In this case, we want the neural network output to have four units, where $n^{[L]} = 4$, like below:

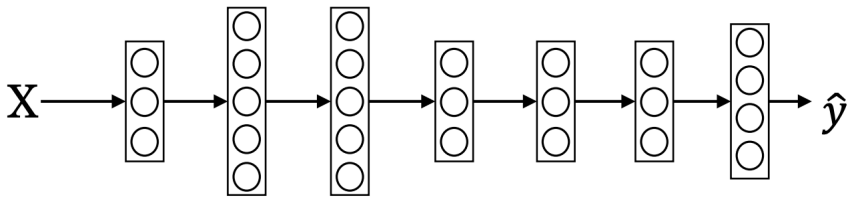

Figure 9.2: Neural Network for Multi-class Classification

We want the number in the output layer would be a $(4, 1)$ vector that tells us the probability of each of four classes in this example.

- The first node would tell us $P(other|x)$.
- The second node would tell us $P(cat|x)$.
- The third node would tell us $P(dog|x)$.
- The fourth node would tell us $P(babyChick|x)$.

Since the probability should sum to 1, the four numbers in the output layer should also sum to 1. The standard model for the neural network to do this uses **softmax layer**.

In the output layer, we are going to compute the linear part of the layer as follows:

$$z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}.$$

Now, we need to apply the softmax activation function as follows:

$$t = e^{z^{[L]}} \text{ (element-wise)}$$
$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_j}.$$

where t is a $(4, 1)$ vector and $a_j^{[L]} = \frac{t_j}{\sum_{j=1}^{4} t_j}$.

Suppose $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$. Then, we have

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}, \sum_{j=1}^{4} t_j = 176.3.$$

Therefore,

$$a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix},$$

which represents the chance of being each of the four classes.

More softmax examples are shown below, where decision boundaries are quite linear:

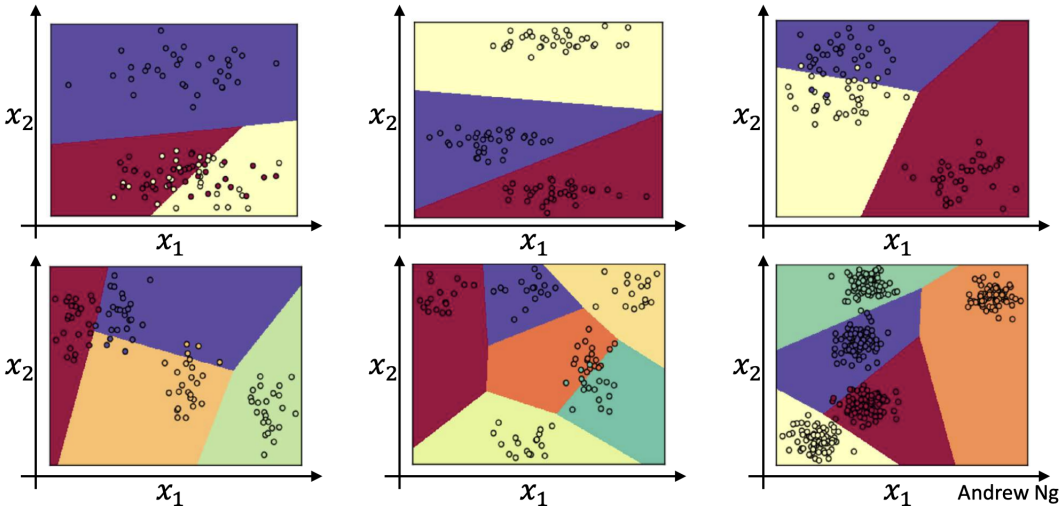

Figure 9.3: Softmax Classification Decision Boundaries without Hidden Layers

9.2 Training a Softmax Classifier

The name softmax comes from contrasting to **hardmax**, where it maps $z^{[L]}$ to $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ in our previous example, where it puts 1 at the position of the biggest element and 0's everywhere else.

Now, let's see how we train the neural network with softmax output layer. Suppose the target output is $y = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$. This means that the input image is a cat image. Further suppose that $a^{[L]} = \hat{y} = \begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$, which is not doing well. In this case, the loss we use is

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log(\hat{y})_j.$$

In our example, $y_1 = y_3 = y_4 = 0, y_2 = 1$. Therefore, the only term left in the loss function is

$$-y_2 \log(\hat{y}_2) = -\log(\hat{y}_2).$$

The way to make loss small is to make $-\log(\hat{y}_2)$ small, which is equivalent of making $\hat{y}_2$ as big as possible, but no bigger than 1. This is in a form of maximum likelihood estimation.

Then, the cost function is defined as follows:

$$J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

When we do backprop, we have gradient of $dz^{[L]}$ as $dz^{[L]} = \hat{y} - y$. The programming framework would take care of gradients for us in this case.

Chapter 10

Introduction to Programming Frameworks

10.1 Deep Learning Frameworks

As we implement more complex neural network models, such as convolutional neural networks or recurrent nerual networks, or we start to build very large models, it is increasingly unpractical to implement everything from scratch. There are many deep learning software frameworks that help us implement these.

The leading deep learning frameworks are outlined below:

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Criteria to choose frameworks:

- Ease of programming (development and deployment) for neural networks.
- Running speed.
- Truly open (open source with good governance) for a long time.

10.2 TensorFlow

As a motivating problem, let's say we want to minimize a cost function

$$J(w) = w^2 - 10w + 25 = (w - 5)^2.$$

The value of w that minimizes the cost function is $w = 5$. Suppose we do not know the value of optimal w, which is often the case when we have multiple parameters, then we can find the optimal parameters using TensorFlow.

First, we need to import packages as follow:

```
import numpy as np
import tensorflow as tf
```

The good thing about TensorFlow is that we only need to implement forward-prop. TensorFlow will figure out how to do backprop, or the gradient computations by Gradient Tape.

We can find the optimal value of w as follows:

```
w = tf.Variable(0, dtype=tf.float32)
optimizer = tf.keras.optimizers.legacy.Adam(learning_rate=0.1)

def train_step():
    with tf.GradientTape() as tape:
        cost = w ** 2 - 10 * w + 25
    trainable_variables = [w]
    grads = tape.gradient(cost, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))
```

After running for 1000 iterations, we obtain $w = 5.000001$ as

$$< tf.Variable'Variable : 0'shape = ()dtype = float32, numpy = 5.000001 > .$$

Suppose the cost function not only depends on parameters but also depends on the training set. We have another version of implementation as follows:

```
w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optimizers.legacy.Adam(learning_rate=0.1)

def training(x, w, optimizer):
    def cost_fn():
        return x[0] * w ** 2 + x[1] * w + x[2]
    for i in range(1000):
        optimizer.minimize(cost_fn, [w])

    return w
```

Similarly, after 1000 iterations, we obtain $w = 5.000001$.