CS 230: Deep Learning

Hargen Zheng

December 23, 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?

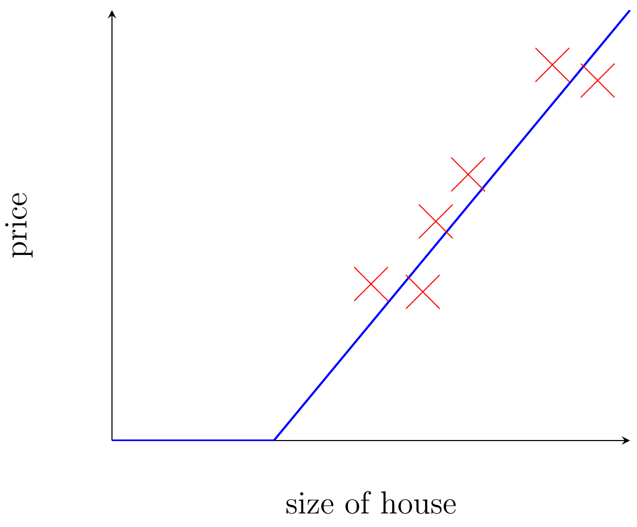

Figure 1.1: Housing Price Prediction

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

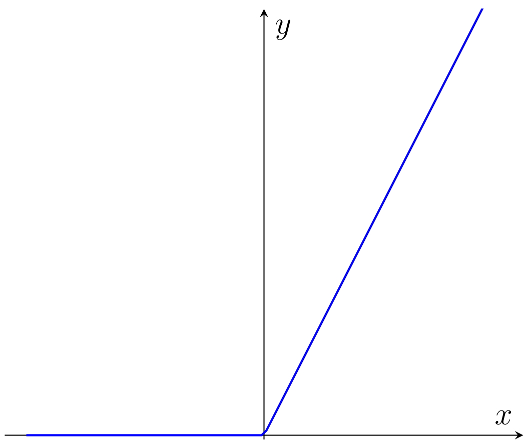

Figure 1.2: ReLU function: $\max\{0, x\}$

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.

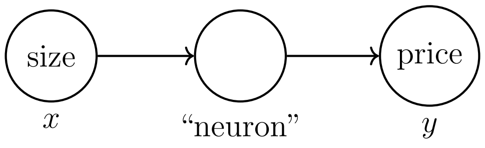

Figure 1.3: Simple Neural Network of Housing Example

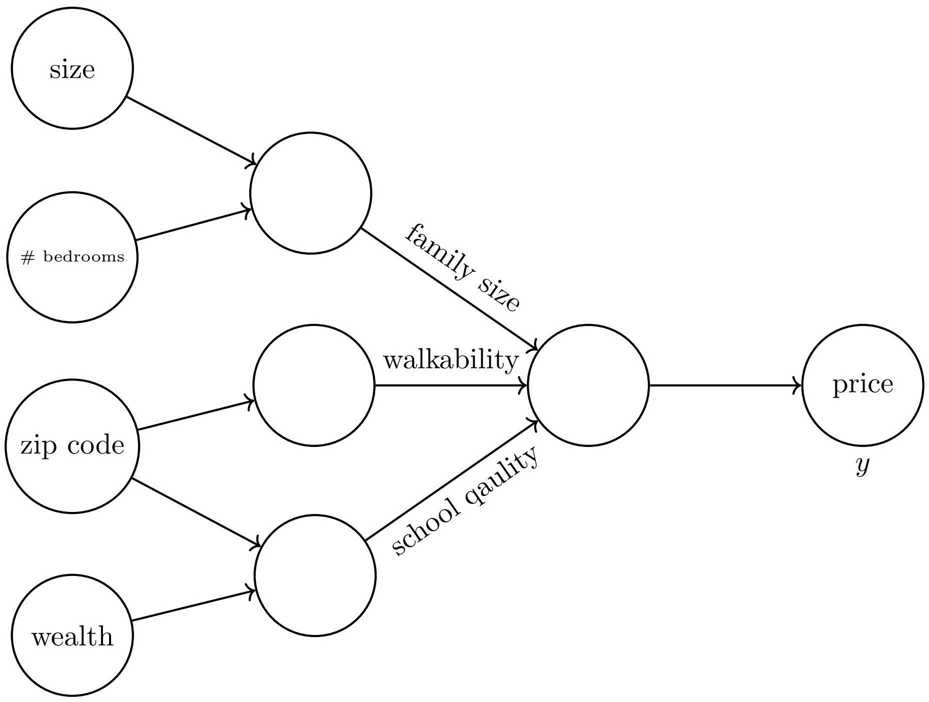

Figure 1.4: Slightly Larger Neural Network

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.

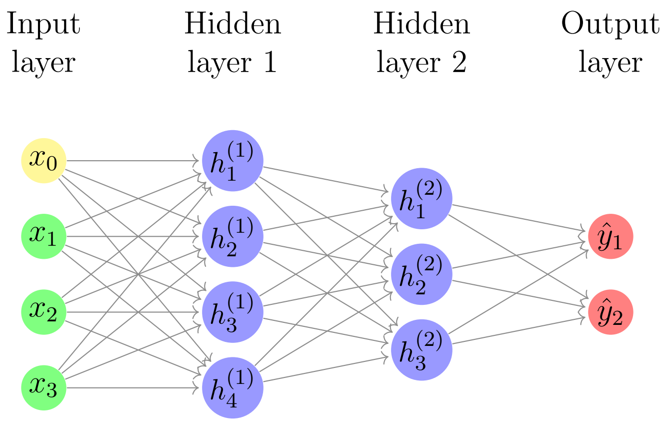

Figure 1.5: General Picture of Neural Network

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

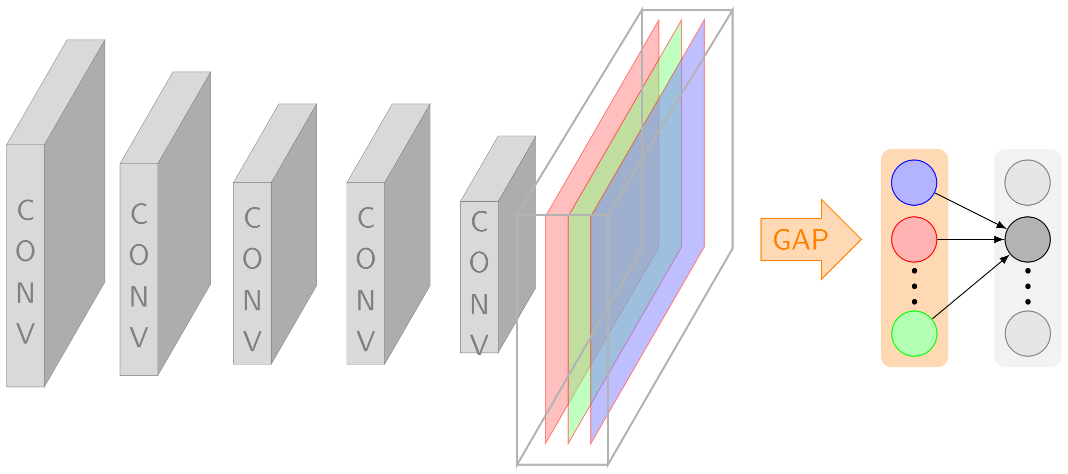

Figure 1.6: Convolutional Neural Network

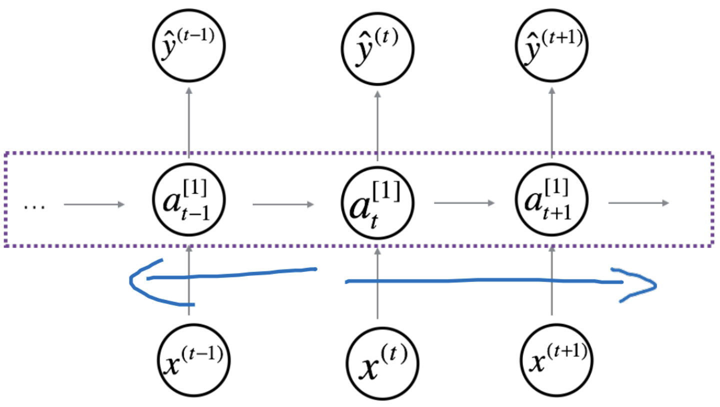

Figure 1.7: Recurrent Neural Network

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

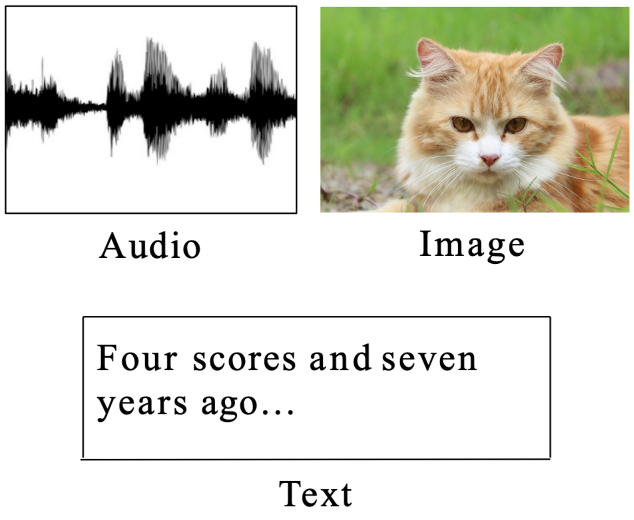

Figure 1.8: Unstructured Data

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?

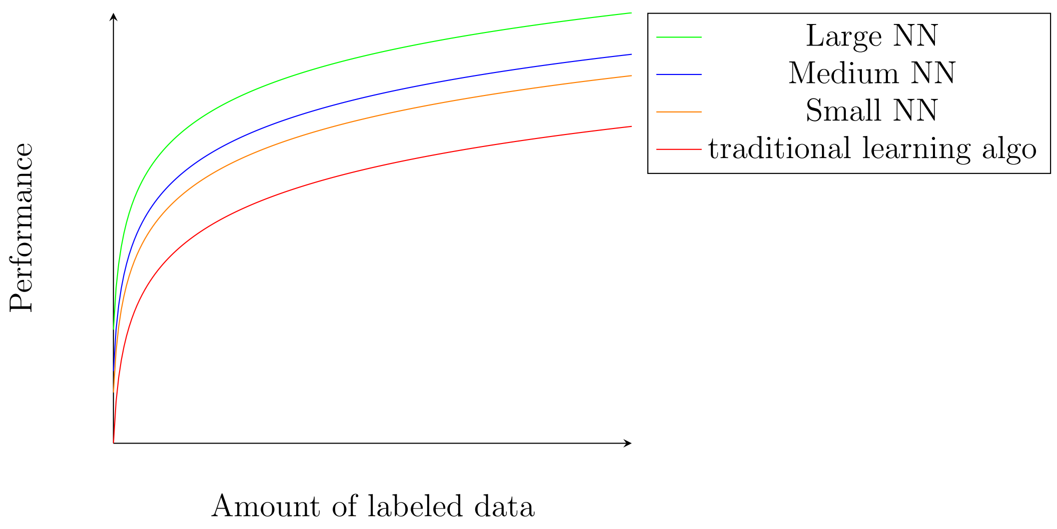

Figure 1.9: Scale drives deep learning progress

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:

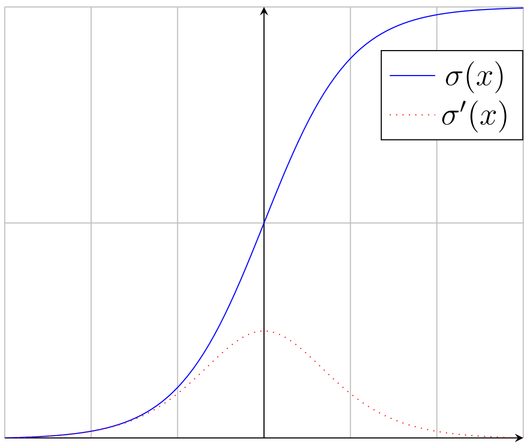

Figure 2.2: Sigmoid Function

2.1.4 Logistic Regression Cost Function

Logistic regression is a supervised learning algorithm, where given training examples $\{(x^{(1)}, y^{(1)}, \dots, (x^{(m)}, y^{(m)})\}$, 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.

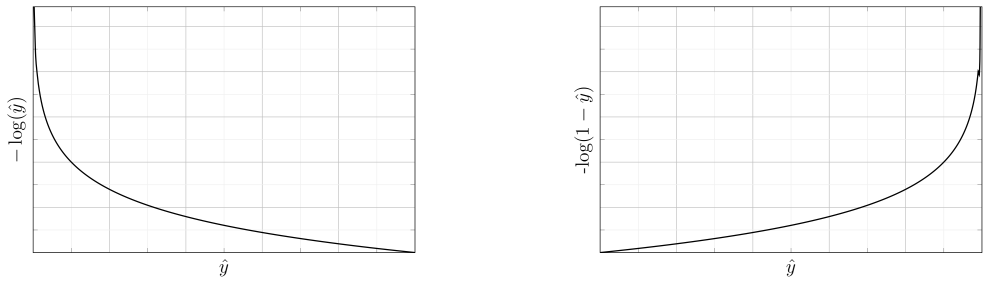

Figure 2.3: A binary classifier's loss function based on if $y = 1$ or $y = 0$.

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.

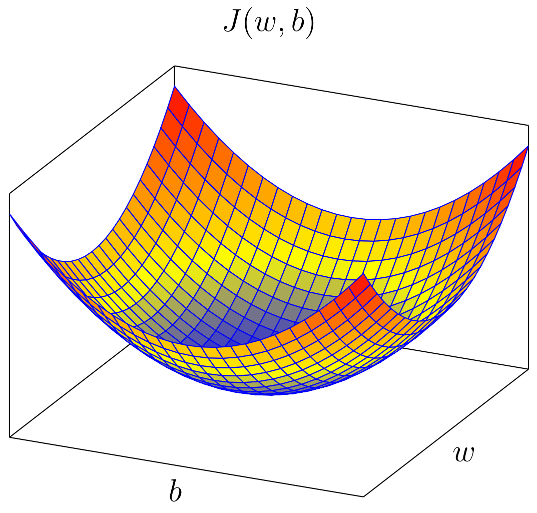

Figure 2.4: Convex Cost Function

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:

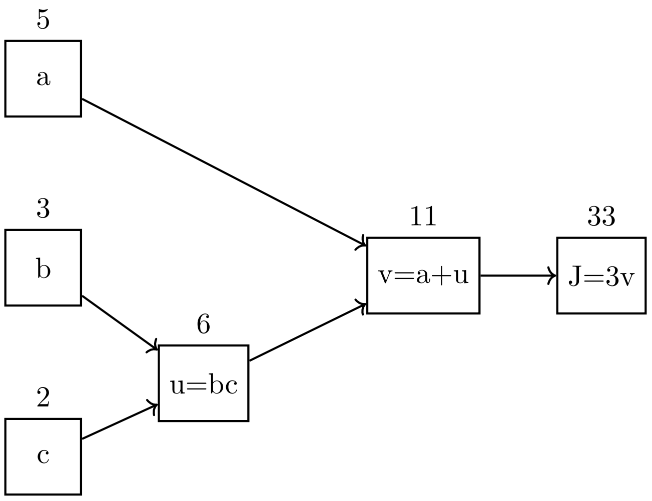

Figure 2.5: Computation Graph for $J(a, b, c) = 3(a + bc)$

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:

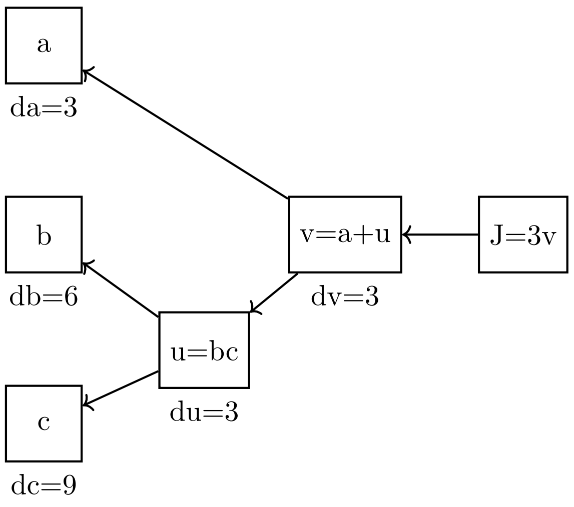

Figure 2.6: Backward Propagation for $J(a, b, c) = 3(a + bc)$

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:

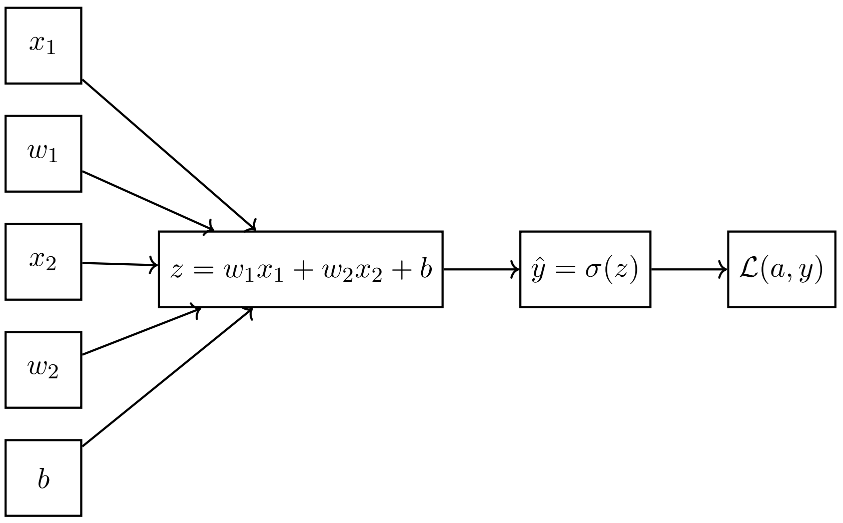

Figure 2.7: Computation Graph for Logistic Regression

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

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

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:

$$\begin{array}{r c} & \begin{array}{cccc} \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \end{array} \\ \begin{array}{r} \text{Carb} \\ \text{Protein} \\ \text{Fat} \end{array} & \begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix} \end{array}$$

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.

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

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.

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

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

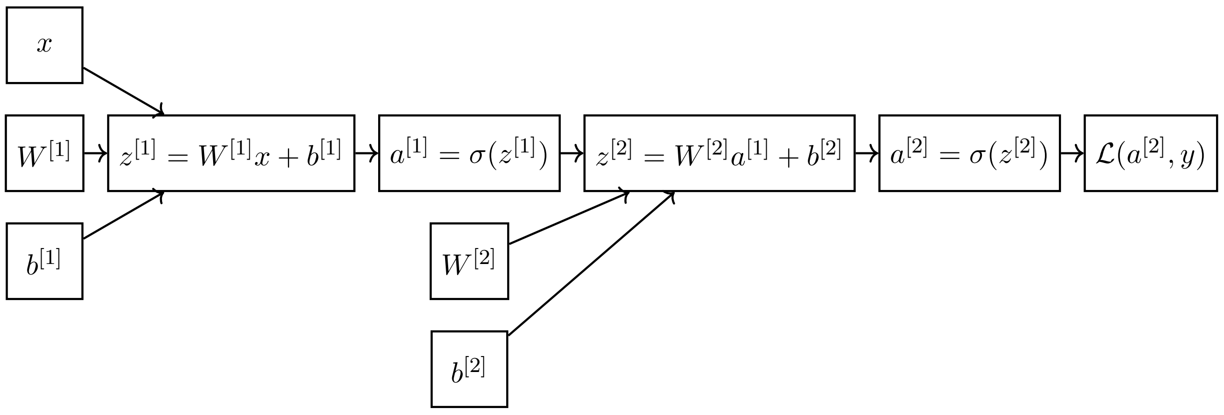
x
$W^{[1]}$
$b^{[1]}$
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$W^{[2]}$
$b^{[2]}$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$\mathcal{L}(a^{[2]}, y)$

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

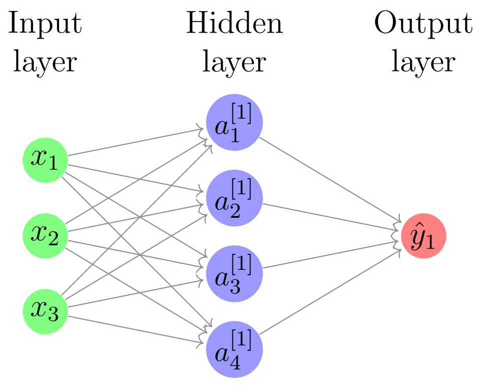

Figure 3.2: Neural Network Representation

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:

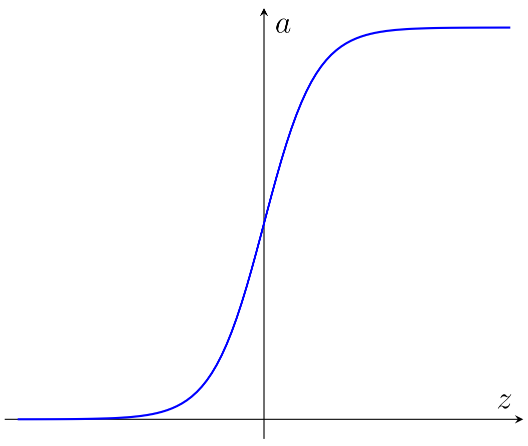

Figure 3.3: Sigmoid Function

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:

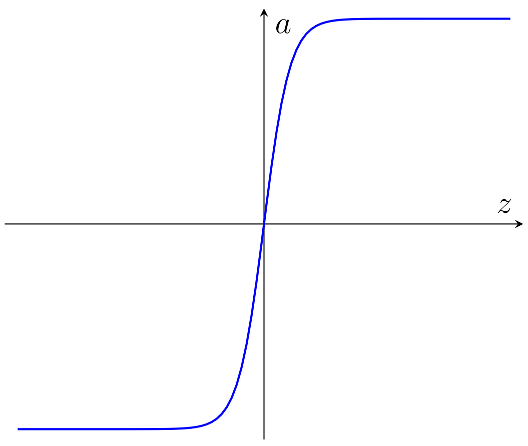

Figure 3.4: Hyperbolic Tangent Function

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

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

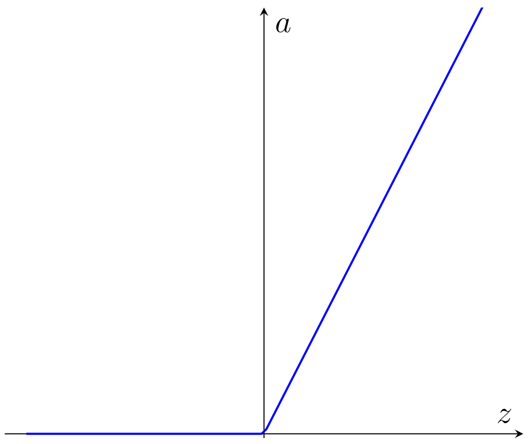

Figure 3.5: ReLU Function

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:

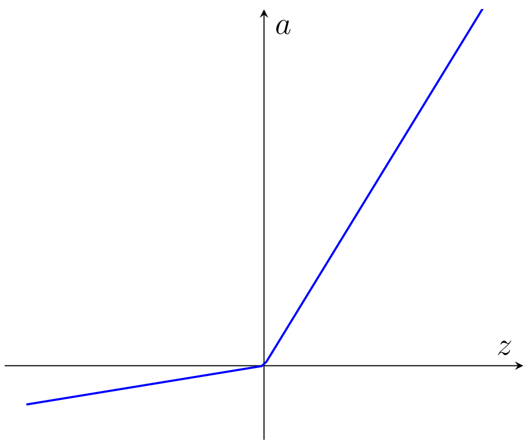

Figure 3.6: Leaky ReLU Function

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:

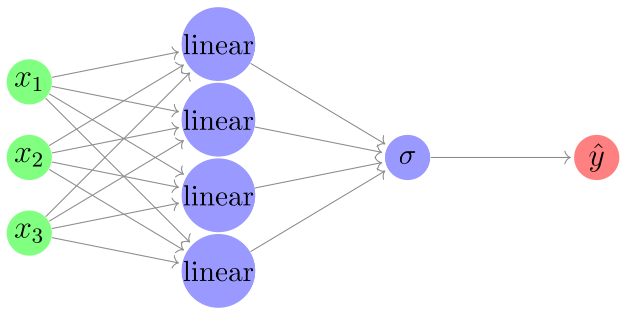

Figure 3.7: Neural Network with Linear Layers

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 $i = 1, \dots, m$.
    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:

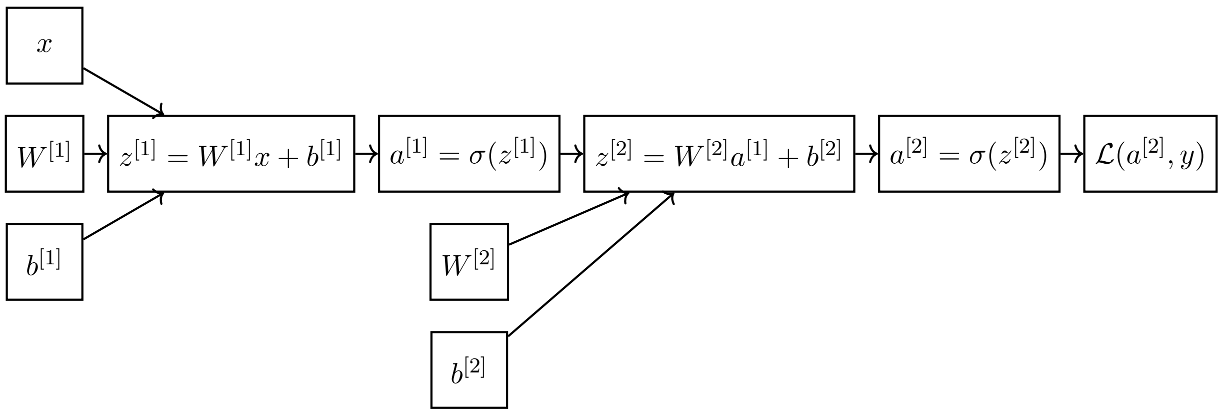
x
$W^{[1]}$
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$\mathcal{L}(a^{[2]}, y)$
$b^{[1]}$
$W^{[2]}$
$b^{[2]}$

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

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

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:

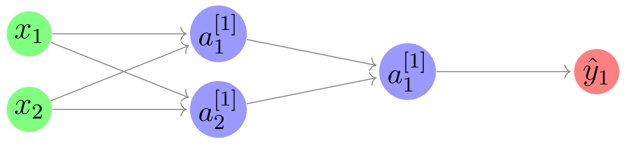

Figure 3.9: Shallow Neural Network Example

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.