XCS229i Lecture Notes

Andrew Ng

Deep Learning

We now begin our study of deep learning. In this set of notes, we give an overview of neural networks, discuss vectorization and discuss training neural networks with backpropagation.

1 Neural Networks

We will start small and slowly build up a neural network, step by step. Recall the housing price prediction problem from before: given the size of the house, we want to predict the price.

Previously, we fitted a straight line to the graph. Now, instead of fitting a straight line, we wish prevent negative housing prices by setting the absolute minimum price as zero. This produces a "kink" in the graph as shown in Figure 1.

Our goal is to input some input x into a function $f(x)$ that outputs the price of the house y. Formally, $f : x \to y$. One of the simplest possible neural networks is to define $f(x)$ as a single "neuron" in the network where $f(x) = \max(ax + b, 0)$, for some coefficients a, b. What $f(x)$ does is return a single value: $(ax + b)$ or zero, whichever is greater. In the context of neural networks, this function is called a ReLU (pronounced "ray-lu"), or rectified linear unit. A more complex neural network may take the single neuron described above and "stack" them together such that one neuron passes its output as input into the next neuron, resulting in a more complex function.

Let us now deepen the housing prediction example. In addition to the size of the house, suppose that you know the number of bedrooms, the zip code

Scribe: Albert Haque

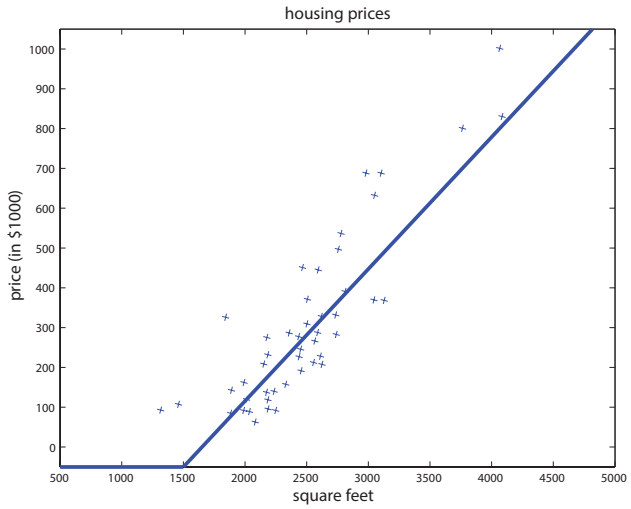


Figure 1: Housing prices with a "kink" in the graph.

and the wealth of the neighborhood. Building neural networks is analogous to Lego bricks: you take individual bricks and stack them together to build complex structures. The same applies to neural networks: we take individual neurons and stack them together to create complex neural networks.

Given these features (size, number of bedrooms, zip code, and wealth), we might then decide that the price of the house depends on the maximum family size it can accommodate. Suppose the family size is a function of the size of the house and number of bedrooms (see Figure 2). The zip code may provide additional information such as how walkable the neighborhood is (i.e., can you walk to the grocery store or do you need to drive everywhere). Combining the zip code with the wealth of the neighborhood may predict the quality of the local elementary school. Given these three derived features (family size, walkable, school quality), we may conclude that the price of the home ultimately depends on these three features.

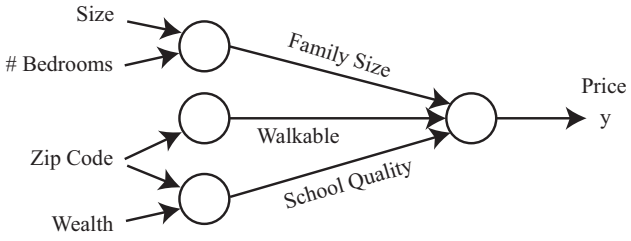


Figure 2: Diagram of a small neural network for predicting housing prices.

We have described this neural network as if you (the reader) already have the insight to determine these three factors ultimately affect the housing price. Part of the magic of a neural network is that all you need are the input features x and the output y while the neural network will figure out everything in the middle by itself. The process of a neural network learning the intermediate features is called *end-to-end learning*.

Following the housing example, formally, the input to a neural network is a set of input features x_1, x_2, x_3, x_4. We connect these four features to three neurons. These three "internal" neurons are called *hidden units*. The goal for the neural network is to automatically determine three relevant features such that the three features predict the price of a house. The only thing we must provide to the neural network is a sufficient number of training examples $(x^{(i)}, y^{(i)})$. Often times, the neural network will discover complex features which are very useful for predicting the output but may be difficult for a human to understand since it does not have a "common" meaning. This is why some people refer to neural networks as a *black box*, as it can be difficult to understand the features it has invented.

Let us formalize this neural network representation. Suppose we have three input features x_1, x_2, x_3 which are collectively called the *input layer*, four hidden units which are collectively called the *hidden layer* and one output neuron called the *output layer*. The term hidden layer is called "hidden" because we do not have the ground truth/training value for the hidden units. This is in contrast to the input and output layers, both of which we know the ground truth values from $(x^{(i)}, y^{(i)})$.

The first hidden unit requires the input x_1, x_2, x_3 and outputs a value denoted by a_1. We use the letter a since it refers to the neuron's "activation" value. In this particular example, we have a single hidden layer but it is possible to have multiple hidden layers. Let $a_1^{[1]}$ denote the output value of the first hidden unit in the first hidden layer. We use zero-indexing to refer to the layer numbers. That is, the input layer is layer 0, the first hidden layer is layer 1 and the output layer is layer 2. Again, more complex neural networks may have more hidden layers. Given this mathematical notation, the output of layer 2 is $a_1^{[2]}$. We can unify our notation:

$$x_1 = a_1^{[0]} \tag{1.1}$$

$$x_2 = a_2^{[0]} \tag{1.2}$$

$$x_3 = a_3^{[0]} \tag{1.3}$$

To clarify, $\text{foo}^{[1]}$ with brackets denotes anything associated with layer 1, $x^{(i)}$ with parenthesis refers to the i^{th} training example, and $a_j^{[\ell]}$ refers to the

activation of the j^{th} unit in layer ℓ. If we look at logistic regression $g(x)$ as a single neuron (see Figure 3):

$$g(x) = \frac{1}{1 + \exp(-w^T x)}$$

The input to the logistic regression $g(x)$ is three features x_1, x_2 and x_3 and it outputs an estimated value of y. We can represent $g(x)$ with a single neuron in the neural network. We can break $g(x)$ into two distinct computations: (1) $z = w^T x + b$ and (2) $a = \sigma(z)$ where $\sigma(z) = 1/(1 + e^{-z})$. Note the notational difference: previously we used $z = \theta^T x$ but now we are using $z = w^T x + b$, where w is a vector. Later in these notes you will see capital W to denote a matrix. The reasoning for this notational difference is conform with standard neural network notation. More generally, $a = g(z)$ where $g(z)$ is some activation function. Example activation functions include:

$$g(z) = \frac{1}{1 + e^{-z}} \qquad \text{(sigmoid)} \tag{1.4}$$

$$g(z) = \max(z, 0) \qquad \text{(ReLU)} \tag{1.5}$$

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \text{(tanh)} \tag{1.6}$$

In general, $g(z)$ is a non-linear function.

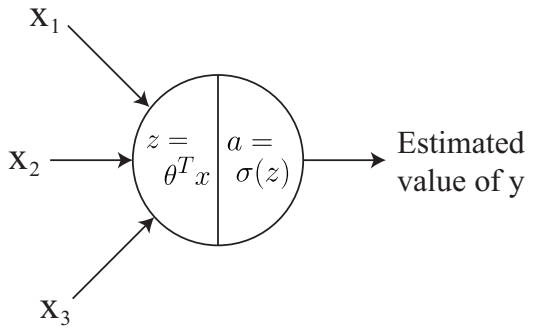


Figure 3: Logistic regression as a single neuron.

Returning to our neural network from before, the first hidden unit in the first hidden layer will perform the following computation:

$$z_1^{[1]} = {W_1^{[1]}}^T x + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \tag{1.7}$$

where W is a matrix of parameters and W_1 refers to the first row of this matrix. The parameters associated with the first hidden unit is the vector

$W_1^{[1]} \in \mathbb{R}^3$ and the scalar $b_1^{[1]} \in \mathbb{R}$. For the second and third hidden units in the first hidden layer, the computation is defined as:

$$z_2^{[1]} = {W_2^{[1]}}^T x + b_2^{[1]} \quad \text{and} \quad a_2^{[1]} = g(z_2^{[1]})$$
$$z_3^{[1]} = {W_3^{[1]}}^T x + b_3^{[1]} \quad \text{and} \quad a_3^{[1]} = g(z_3^{[1]})$$

where each hidden unit has its corresponding parameters W and b. Moving on, the output layer performs the computation:

$$z_1^{[2]} = {W_1^{[2]}}^T a^{[1]} + b_1^{[2]} \quad \text{and} \quad a_1^{[2]} = g(z_1^{[2]}) \tag{1.8}$$

where $a^{[1]}$ is defined as the concatenation of all first layer activations:

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \tag{1.9}$$

The activation $a_1^{[2]}$ from the second layer, which is a single scalar as defined by $a_1^{[2]} = g(z_1^{[2]})$, represents the neural network's final output prediction. Note that for regression tasks, one typically does not apply a non-linear function which is strictly positive (i.e., ReLU or sigmoid) because for some tasks, the ground truth y value may in fact be negative.

2 Vectorization

In order to implement a neural network at a reasonable speed, one must be careful when using for loops. In order to compute the hidden unit activations in the first layer, we must compute $z_1, ..., z_4$ and $a_1, ..., a_4$.

$$z_1^{[1]} = {W_1^{[1]}}^T x + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \tag{2.1}$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \tag{2.2}$$
$$z_4^{[1]} = {W_4^{[1]}}^T x + b_4^{[1]} \quad \text{and} \quad a_4^{[1]} = g(z_4^{[1]}) \tag{2.3}$$

The most natural way to implement this in code is to use a for loop. One of the treasures that deep learning has given to the field of machine learning is that deep learning algorithms have high computational requirements. As a result, code will run very slowly if you use for loops.

This gave rise to *vectorization*. Instead of using for loops, vectorization takes advantage of matrix algebra and highly optimized numerical linear algebra packages (e.g., BLAS) to make neural network computations run quickly. Before the deep learning era, a for loop may have been sufficient on smaller datasets, but modern deep networks and state-of-the-art datasets will be infeasible to run with for loops.

2.1 Vectorizing the Output Computation

We now present a method for computing $z_1, ..., z_4$ without a for loop. Using our matrix algebra, we can compute the activations:

$$\underbrace{\begin{bmatrix} z_1^{[1]} \\ \vdots \\ \vdots \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]} \in \mathbb{R}^{4\times 1}} = \underbrace{\begin{bmatrix} — \; {W_1^{[1]}}^T \; — \\ — \; {W_2^{[1]}}^T \; — \\ \vdots \\ — \; {W_4^{[1]}}^T \; — \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{4\times 3}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x \in \mathbb{R}^{3\times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{4\times 1}} \tag{2.4}$$

Where the $\mathbb{R}^{d\times n}$ beneath each matrix indicates the dimensions. Expressing this in matrix notation: $z^{[1]} = W^{[1]}x + b^{[1]}$. To compute $a^{[1]}$ without a for loop, we can leverage vectorized libraries in Matlab, Octave, or Python which compute $a^{[1]} = g(z^{[1]})$ very fast by performing parallel element-wise operations. Mathematically, we defined the sigmoid function $g(z)$ as:

$$g(z) = \frac{1}{1+e^{-z}} \qquad \text{where } z \in \mathbb{R} \tag{2.5}$$

However, the sigmoid function can be defined not only for scalars but also vectors. In a Matlab/Octave-like pseudocode, we can define the sigmoid as:

$$g(z) = \texttt{1 ./ (1+exp(-}z\texttt{))} \qquad \text{where } z \in \mathbb{R}^d \tag{2.6}$$

where `./` denotes element-wise division. With this vectorized implementation, $a^{[1]} = g(z^{[1]})$ can be computed quickly.

To summarize the neural network so far, given an input $x \in \mathbb{R}^3$, we compute the hidden layer's activations with $z^{[1]} = W^{[1]}x + b^{[1]}$ and $a^{[1]} = g(z^{[1]})$. To compute the output layer's activations (i.e., neural network output):

$$\underbrace{z^{[2]}}_{1\times 1} = \underbrace{W^{[2]}}_{1\times 4}\underbrace{a^{[1]}}_{4\times 1} + \underbrace{b^{[2]}}_{1\times 1} \quad \text{and} \quad \underbrace{a^{[2]}}_{1\times 1} = g(\underbrace{z^{[2]}}_{1\times 1}) \tag{2.7}$$

Why do we not use the identity function for $g(z)$? That is, why not use $g(z) = z$? Assume for sake of argument that $b^{[1]}$ and $b^{[2]}$ are zeros. Using Equation (2.7), we have:

$$\begin{aligned} z^{[2]} &= W^{[2]}a^{[1]} && && (2.8)\\ &= W^{[2]}g(z^{[1]}) && \text{by definition} && (2.9)\\ &= W^{[2]}z^{[1]} && \text{since } g(z) = z && (2.10)\\ &= W^{[2]}W^{[1]}x && \text{from Equation (2.4)} && (2.11)\\ &= \tilde{W}x && \text{where } \tilde{W} = W^{[2]}W^{[1]} && (2.12) \end{aligned}$$

Notice how $W^{[2]}W^{[1]}$ collapsed into $\tilde{W}$. This is because applying a linear function to another linear function will result in a linear function over the original input (i.e., you can construct a $\tilde{W}$ such that $\tilde{W}x = W^{[2]}W^{[1]}x$). This loses much of the representational power of the neural network as often times the output we are trying to predict has a non-linear relationship with the inputs. Without non-linear activation functions, the neural network will simply perform linear regression.

2.2 Vectorization Over Training Examples

Suppose you have a training set with three examples. The activations for each example are as follows:

$$\begin{aligned} z^{[1](1)} &= W^{[1]}x^{(1)} + b^{[1]}\\ z^{[1](2)} &= W^{[1]}x^{(2)} + b^{[1]}\\ z^{[1](3)} &= W^{[1]}x^{(3)} + b^{[1]} \end{aligned}$$

Note the difference between square brackets $[\cdot]$, which refer to the layer number, and parenthesis $(\cdot)$, which refer to the training example number. Intuitively, one would implement this using a for loop. It turns out, we can vectorize these operations as well. First, define:

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \tag{2.13}$$

Note that we are stacking training examples in columns and *not* rows. We can then combine this into a single unified formulation:

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]} \tag{2.14}$$

You may notice that we are attempting to add $b^{[1]} \in \mathbb{R}^{4\times 1}$ to $W^{[1]}X \in \mathbb{R}^{4\times 3}$. Strictly following the rules of linear algebra, this is not allowed. In practice however, this addition is performed using *broadcasting*. We create an intermediate $\tilde{b}^{[1]} \in \mathbb{R}^{4\times 3}$:

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \tag{2.15}$$

We can then perform the computation: $Z^{[1]} = W^{[1]}X + \tilde{b}^{[1]}$. Often times, it is not necessary to explicitly construct $\tilde{b}^{[1]}$. By inspecting the dimensions in (2.14), you can assume $b^{[1]} \in \mathbb{R}^{4\times 1}$ is correctly broadcast to $W^{[1]}X \in \mathbb{R}^{4\times 3}$.

Putting it together: Suppose we have a training set $(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})$ where $x^{(i)}$ is a picture and $y^{(i)}$ is a binary label for whether the picture contains a cat or not (i.e., 1=contains a cat). First, we initialize the parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ to small random numbers. For each example, we compute the output "probability" from the sigmoid function $a^{[2](i)}$. Second, using the logistic regression log likelihood:

$$\sum_{i=1}^{n} \left(y^{(i)} \log a^{[2](i)} + (1 - y^{(i)}) \log(1 - a^{[2](i)}) \right) \tag{2.16}$$

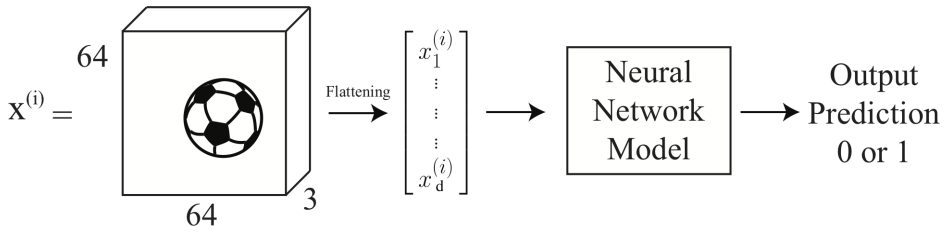
Finally, we maximize this function using gradient ascent. This maximization procedure corresponds to training the neural network.

3 Backpropagation

Instead of the housing example, we now have a new problem. Suppose we wish to detect whether there is a soccer ball in an image or not. Given an input image $x^{(i)}$, we wish to output a binary prediction 1 if there is a ball in the image and 0 otherwise.

Aside: Images can be represented as a matrix with number of elements equal to the number of pixels. However, color images are digitally represented as a volume (i.e., three-channels; or three matrices stacked on each other). The number three is used because colors are represented as red-green-blue (RGB) values. In the diagram below, we have a $64 \times 64 \times 3$ image containing a soccer ball. It is *flattened* into a single vector containing 12,288 elements.

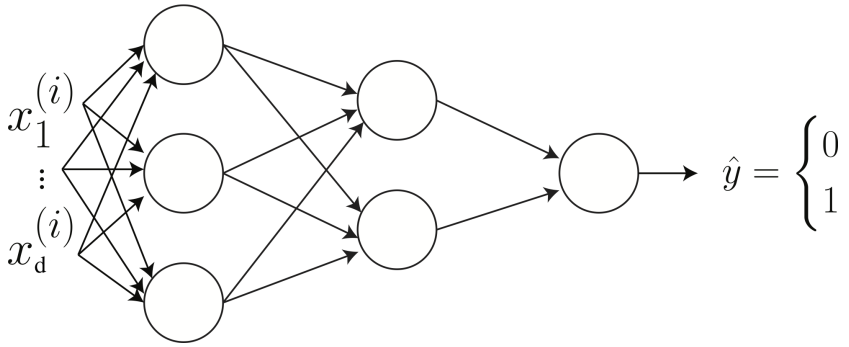
A neural network *model* consists of two components: (i) the network architecture, which defines how many layers, how many neurons, and how the neurons are connected and (ii) the parameters (values; also known as



weights). In this section, we will talk about how to learn the parameters. First we will talk about parameter initialization, optimization and analyzing these parameters.

3.1 Parameter Initialization

Consider a two layer neural network. On the left, the input is a flattened image vector $x^{(1)}, ..., x_d^{(i)}$. In the first hidden layer, notice how all inputs are connected to all neurons in the next layer. This is called a *fully connected* layer.



The next step is to compute how many parameters are in this network. One way of doing this is to compute the forward propagation by hand.

$$z^{[1]} = W^{[1]}x^{(i)} + b^{[1]} \tag{3.1}$$
$$a^{[1]} = g(z^{[1]}) \tag{3.2}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{3.3}$$
$$a^{[2]} = g(z^{[2]}) \tag{3.4}$$
$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]} \tag{3.5}$$
$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]}) \tag{3.6}$$

We know that $z^{[1]}, a^{[1]} \in \mathbb{R}^{3\times 1}$ and $z^{[2]}, a^{[2]} \in \mathbb{R}^{2\times 1}$ and $z^{[3]}, a^{[3]} \in \mathbb{R}^{1\times 1}$. As of now, we do not know the size of $W^{[1]}$. However, we can compute its size.

We know that $x \in \mathbb{R}^{d \times 1}$. This leads us to the following

$$z^{[1]} = W^{[1]}x^{(i)} = \mathbb{R}^{3\times 1} \quad \text{Written as sizes:} \quad \mathbb{R}^{3\times 1} = \mathbb{R}^{?\times ?} \times \mathbb{R}^{d\times 1} \tag{3.7}$$

Using matrix multiplication, we conclude that $?\times?$ must be $3 \times d$. We also conclude that the bias is of size 3×1 because its size must match $W^{[1]}x^{(i)}$. We repeat this process for each hidden layer. This gives us:

$$W^{[2]} \in \mathbb{R}^{2\times 3}, b^{[2]} \in \mathbb{R}^{2\times 1} \quad \text{and} \quad W^{[3]} \in \mathbb{R}^{1\times 2}, b^{[3]} \in \mathbb{R}^{1\times 1} \tag{3.8}$$

In total, we have $3d + 3$ in the first layer, $2 \times 3 + 2$ in the second layer and $2 + 1$ in the third layer. This gives us a total of $3d + 14$ parameters.

Before we start training the neural network, we must select an initial value for these parameters. We do not use the value zero as the initial value. This is because the output of the first layer will always be the same since $W^{[1]}x^{(i)} + b^{[1]} = 0^{3\times 1}x^{(i)} + 0^{3\times 1}$ where $0^{d\times n}$ denotes a matrix of size $d \times n$ filled with zeros. This will cause problems later on when we try to update these parameters (i.e., the gradients will all be the same). The solution is to randomly initialize the parameters to small values (e.g., normally distributed around zero; $\mathcal{N}(0, 0.1)$). Once the parameters have been initialized, we can begin training the neural network with gradient descent.

The next step of the training process is to update the parameters. After a single forward pass through the neural network, the output will be a predicted value $\hat{y}$. We can then compute the loss $\mathcal{L}$, in our case the log loss:

$$\mathcal{L}(\hat{y}, y) = -\Big[(1 - y)\log(1 - \hat{y}) + y \log \hat{y}\Big] \tag{3.9}$$

The loss function $\mathcal{L}(\hat{y}, y)$ produces a single scalar value. For short, we will refer to the loss value as $\mathcal{L}$. Given this value, we now must update all parameters in layers of the neural network. For any given layer index ℓ, we update them:

$$W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial W^{[\ell]}} \tag{3.10}$$

$$b^{[\ell]} = b^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial b^{[\ell]}} \tag{3.11}$$

where α is the learning rate. To proceed, we must compute the gradient with respect to the parameters: $\partial\mathcal{L}/\partial W^{[\ell]}$ and $\partial\mathcal{L}/\partial b^{[\ell]}$.

Remember, we made a decision to not set all parameters to zero. What if we had initialized all parameters to be zero? We know that $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$

will evaluate to zero, because $W^{[3]}$ and $b^{[3]}$ are all zero. However, the output of the neural network is defined as $a^{[3]} = g(z^{[3]})$. Recall that $g(\cdot)$ is defined as the sigmoid function. This means $a^{[3]} = g(0) = 0.5$. Thus, no matter what value of $x^{(i)}$ we provide, the network will output $\hat{y} = 0.5$.

What if we had initialized all parameters to be the same non-zero value? In this case, consider the activations of the first layer:

$$a^{[1]} = g(z^{[1]}) = g(W^{[1]}x^{(i)} + b^{[1]}) \tag{3.12}$$

Each element of the activation vector $a^{[1]}$ will be the same (because $W^{[1]}$ contains all the same values). This behavior will occur at all layers of the neural network. As a result, when we compute the gradient, all neurons in a layer will be equally responsible for anything contributed to the final loss. We call this property *symmetry*. This means each neuron (within a layer) will receive the exact same gradient update value (i.e., all neurons will learn the same thing).

In practice, it turns out there is something better than random initialization. It is called Xavier/He initialization and initializes the weights:

$$w^{[\ell]} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n^{[\ell]} + n^{[\ell-1]}}}\right) \tag{3.13}$$

where $n^{[\ell]}$ is the number of neurons in layer ℓ. This acts as a mini-normalization technique. For a single layer, consider the variance of the input to the layer as $\sigma^{(in)}$ and the variance of the output (i.e., activations) of a layer to be $\sigma^{(out)}$. Xavier/He initialization encourages $\sigma^{(in)}$ to be similar to $\sigma^{(out)}$.

3.2 Optimization

Recall our neural network parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}$. To update them, we use stochastic gradient descent (SGD) using the update rules in Equations (3.10) and (3.11). So our goal is to calculate $\frac{\partial \mathcal{L}}{\partial W^{[1]}}$, $\frac{\partial \mathcal{L}}{\partial W^{[2]}}$, $\frac{\partial \mathcal{L}}{\partial W^{[3]}}$, $\frac{\partial \mathcal{L}}{\partial b^{[1]}}$, $\frac{\partial \mathcal{L}}{\partial b^{[2]}}$ and $\frac{\partial \mathcal{L}}{\partial b^{[3]}}$. In what follows we will compute the gradient with respect to $W^{[2]}$ and leave the rest as an exercise since they are very similar.

First, observe that

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial W^{[2]}_{11}} & \frac{\partial \mathcal{L}}{\partial W^{[2]}_{12}} & \frac{\partial \mathcal{L}}{\partial W^{[2]}_{13}} \\ \frac{\partial \mathcal{L}}{\partial W^{[2]}_{21}} & \frac{\partial \mathcal{L}}{\partial W^{[2]}_{22}} & \frac{\partial \mathcal{L}}{\partial W^{[2]}_{23}} \end{bmatrix},$$

and also observe that

$$\frac{\partial \mathcal{L}}{\partial z^{[3]}} = \frac{\partial}{\partial z^{[3]}} \left[-y \log \hat{y} - (1-y)\log(1-\hat{y})\right]$$

$$
\begin{aligned}
&= \frac{\partial}{\partial z^{[3]}} \left[-y \log \sigma(z^{[3]}) - (1-y) \log(1 - \sigma(z^{[3]})) \right] \qquad \text{(where } \sigma \text{ is the sigmoid function)} \\
&= -y \frac{1}{\sigma(z^{[3]})} \sigma(z^{[3]})(1 - \sigma(z^{[3]})) - (1-y) \frac{1}{(1 - \sigma(z^{[3]}))} (-1) \sigma(z^{[3]})(1 - \sigma(z^{[3]})) \\
&= -y(1 - \sigma(z^{[3]}) + (1-y)\sigma(z^{[3]}) \\
&= \sigma(z^{[3]}) - y \\
&= a^{[3]} - y.
\end{aligned}
$$

Now to calculate the gradient w.r.t $\frac{\partial \mathcal{L}}{\partial W_{ij}^{[2]}}$, we use the multivariate chain rule of calculus:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{ij}^{[2]}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_{ij}^{[2]}} \\
&= \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial W_{ij}^{[2]}} \\
&= \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial W_{ij}^{[2]}} \\
&= \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial W_{ij}^{[2]}} \\
&= \underbrace{\frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}}}_{\underbrace{(a^{[3]} - y)}_{1\times 1}} \underbrace{\frac{\partial z^{[3]}}{\partial a^{[2]}}}_{\underbrace{W^{[3]}}_{1\times 2}} \underbrace{\frac{\partial a^{[2]}}{\partial z^{[2]}}}_{\underbrace{\text{diag}\left(g'(z^{[2]})\right)}_{2\times 2}} \underbrace{\frac{\partial z^{[2]}}{\partial W_{ij}^{[2]}}}_{\underbrace{a_j^{[1]} \mathbf{e_i}}_{2\times 1}} \\
&\qquad \text{(where } a^{[1]} \in \mathbb{R}^3 \text{, and } \mathbf{e_i} \in \mathbb{R}^2 \text{ is the } i^{th} \text{ basis vector)} \\
&= \underbrace{(a^{[3]} - y) W^{[3]} \circ g'(z^{[2]})}_{1\times 2} \quad \underbrace{a_j^{[1]} \mathbf{e_i}}_{2\times 1} \\
&= \underbrace{\left[(a^{[3]} - y) W^{[3]} \circ g'(z^{[2]}) \right]_i a_j^{[1]}}_{1\times 1} \\
\Rightarrow \frac{\partial \mathcal{L}}{\partial W^{[2]}} &= \underbrace{\left[(a^{[3]} - y) W^{[3]} \circ g'(z^{[2]}) \right] a^{[1]^T}}_{2\times 3}
\end{aligned}
$$

where $\circ$ indicates elemntwise product (Hadamard product). We leave the remaining gradients as an exercise to the reader.

Returning to optimization, we previously discussed stochastic gradient descent. Now we will talk about gradient descent. For any single layer ℓ, the update rule is defined as:

$$W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial J}{\partial W^{[\ell]}} \tag{3.14}$$

where J is the cost function $J = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}^{(i)}$ and $\mathcal{L}^{(i)}$ is the loss for a single example. The difference between the gradient descent update versus the stochastic gradient descent version is that the cost function J gives more accurate gradients whereas $\mathcal{L}^{(i)}$ may be noisy. Stochastic gradient descent attempts to approximate the gradient from (full) gradient descent. The disadvantage of gradient descent is that it can be difficult to compute all activations for all examples in a single forward or backwards propagation phase.

In practice, research and applications use *mini-batch gradient descent*. This is a compromise between gradient descent and stochastic gradient descent. In the case mini-batch gradient descent, the cost function J_{mb} is defined as follows:

$$J_{\text{mb}} = \frac{1}{\text{B}} \sum_{i=1}^{\text{B}} \mathcal{L}^{(i)} \tag{3.15}$$

where B is the number of examples in the mini-batch.

There is another optimization method called *momentum*. Consider mini-batch stochastic gradient. For any single layer ℓ, the update rule is as follows:

$$\begin{cases} v_{dW^{[\ell]}} = \beta v_{dW^{[\ell]}} + (1 - \beta) \frac{\partial J}{\partial W^{[\ell]}} \\ W^{[\ell]} = W^{[\ell]} - \alpha v_{dW^{[\ell]}} \end{cases} \tag{3.16}$$

Notice how there are now two stages instead of a single stage. The weight update now depends on the cost J at this update step and the *velocity* $v_{dW^{[\ell]}}$. The relative importance is controlled by β. Consider the analogy to a human driving a car. While in motion, the car has momentum. If the car were to use the brakes (or not push accelerator throttle), the car would continue moving due to its momentum. Returning to optimization, the velocity $v_{dW^{[\ell]}}$ will keep track of the gradient over time. This technique has significantly helped neural networks during the training phase.

3.3 Analyzing the Parameters

At this point, we have initialized the parameters and have optimized the parameters. Suppose we evaluate the trained model and observe that it

achieves 96% accuracy on the training set but only 64% on the testing set. Some solutions include: collecting more data, employing regularization, or making the model shallower. Let us briefly look at regularization techniques.

3.3.1 L2 Regularization

Let W below denote *all* the parameters in a model. In the case of neural networks, you may think of applying the 2nd term to all layer weights $W^{[\ell]}$. For convenience, we simply write W. The L2 regularization adds another term to the cost function:

$$J_{L2} = J + \frac{\lambda}{2}||W||^2 \tag{3.17}$$

$$= J + \frac{\lambda}{2}\sum_{ij}|W_{ij}|^2 \tag{3.18}$$

$$= J + \frac{\lambda}{2}W^T W \tag{3.19}$$

where J is the standard cost function from before, λ is an arbitrary value with a larger value indicating more regularization and W contains all the weight matrices, and where Equations (3.17), (3.18) and (3.19) are equivalent. The update rule with L2 regularization becomes:

$$W = W - \alpha\frac{\partial J}{\partial W} - \alpha\frac{\lambda}{2}\frac{\partial W^T W}{\partial W} \tag{3.20}$$

$$= (1 - \alpha\lambda)W - \alpha\frac{\partial J}{\partial W} \tag{3.21}$$

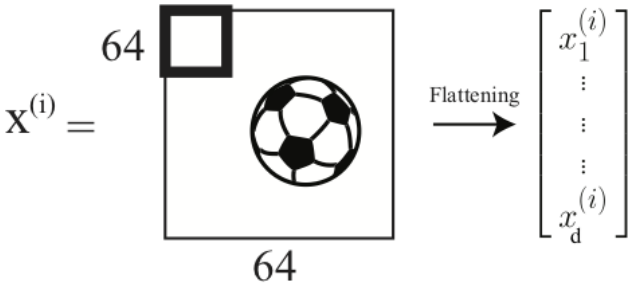
When we were updating our parameters using gradient descent, we did not have the $(1 - \alpha\lambda)W$ term. This means with L2 regularization, every update will include some penalization, depending on W. This penalization increases the cost J, which encourages individual parameters to be small in magnitude, which is a way to reduce overfitting.

3.3.2 Parameter Sharing

Recall logistic regression. It can be represented as a neural network, as shown in Figure 3. The parameter vector $\theta = (\theta_1, ..., \theta_d)$ must have the same number of elements as the input vector $x = (x_1, ..., x_d)$. In our image soccer ball example, this means θ_1 always looks at the top left pixel of the image no matter what. However, we know that a soccer ball might appear in any region of the image and not always the center. It is possible that θ_1 was

never trained on a soccer ball in the top left of the image. As a result, during test time, if an image of a soccer ball in the top left appears, the logistic regression will likely predict *no soccer ball*. This is a problem.

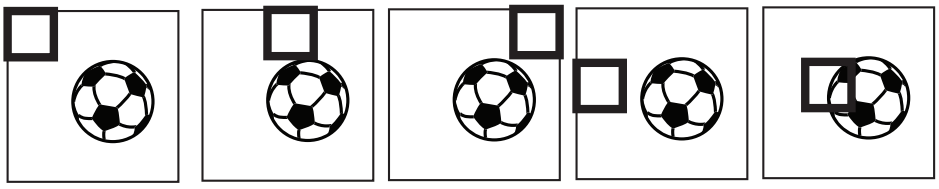
This leads us to *convolutional neural networks*. Suppose θ is no longer a vector but instead is a matrix. For our soccer ball example, suppose $\theta = \mathbb{R}^{4\times 4}$. For simplicity, we show the image as 64×64 but recall it is actually three-



dimensional and contains 3 channels. We now take our matrix of parameters θ and slide it over the image. This is shown above by the thick square in the upper left of the image. To compute the activation a, we compute the element-wise product between θ and $x_{1:4,1:4}$, where the subscripts for x indicate we are taking the top left 4×4 region in the image x. We then collapse the matrix into a single scalar by summing all the elements resulting from the element-wise product. Formally:

$$a = \sum_{i=1}^{4}\sum_{j=1}^{4} \theta_{ij} x_{ij} \tag{3.22}$$

We then move this window slightly to the right in the image and repeat this process. Once we have reached the end of the row, we start at the beginning of the second row.



Once we have reached the end of the image, the parameters θ have "seen" all pixels of the image: θ_1 is no longer related to only the top left pixel. As a result, whether the soccer ball appears in the bottom right or top left of the image, the neural network will successfully detect the soccer ball.