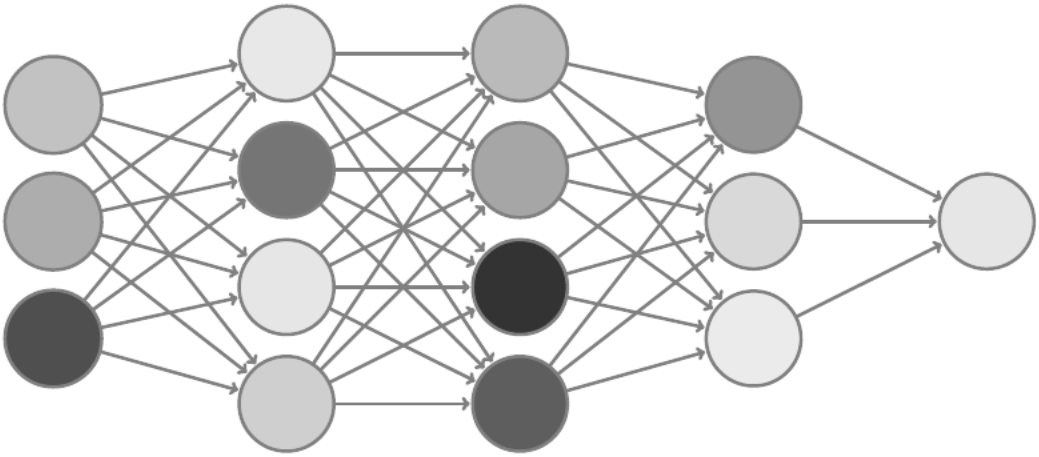
Deep Learning

Stanford CS230

Matt Deitke



Last updated: February 15, 2020

Acknowledgements

This set of notes follows the lectures from Stanford's graduate-level course CS230: Deep Learning. The course was taught by **Andrew Ng**, **Kian Katanforoosh**. The course website is *cs230.stanford.edu*.

Contents

Chapter 1

Introduction

The gist of deep learning, and the algorithms behind it, have been around for decades. However, we saw that as we started to add data to neural networks, they started to perform much better than traditional machine learning algorithms. With advances in GPU computing and the amount of data we have available, training larger neural networks are easier than ever before. With more data, larger neural networks have been shown to outperform all other machine learning algorithms.

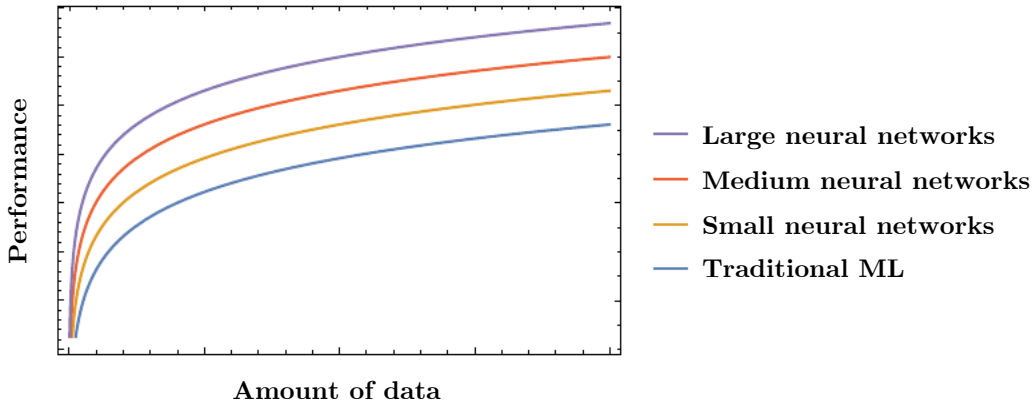
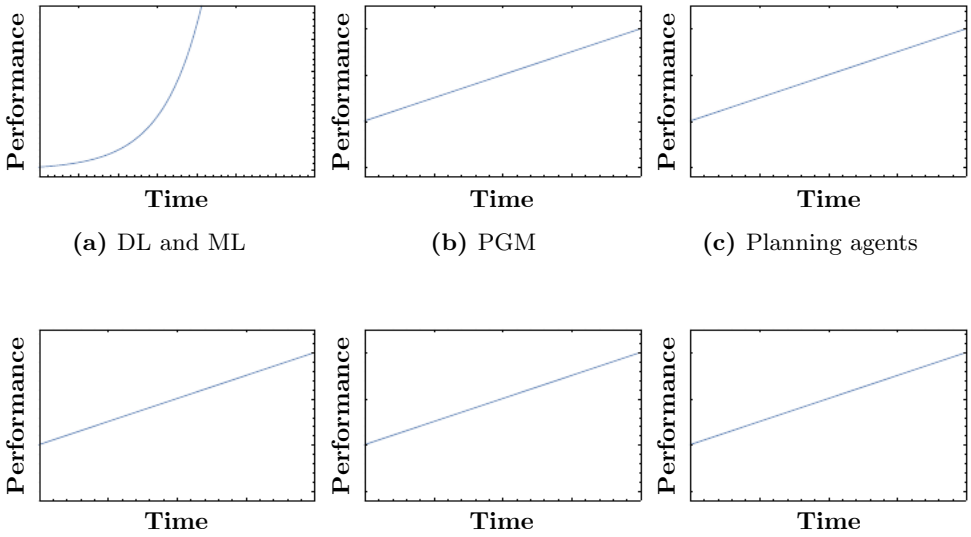


Figure 1.0.1: With more data, neural networks have performed better than traditional machine learning.

Artificial intelligence can be broken down into a few subfields that include deep learning (DL), machine learning (ML), probabilistic graphical models (PGM), planning agents, search algorithms, knowledge representation (KR), and game theory. The only subfield that has dramatically improved in performance has been deep learning and machine learning.

We can see the rise of artificial intelligence in many industries today, however, there is still a long way to go. Even though a company uses neural networks, the company is not an artificial intelligence company. The best AI companies are good at strategic data acquisition, putting data together, spotting automation, and have job descriptions on technologies at the forefront of the field.



(a) DL and ML **(b)** PGM **(c)** Planning agents

(d) Search algorithms **(e)** Knowledge representation **(f)** Game theory

Figure 1.0.2: The performance of deep learning and machine learning algorithms have been exploding in the last few years, in comparison to other branches of artificial intelligence.

1.1 Deep learning

1.1.1 What is a neural network?

The aim of a neural network is to learn representations that best predict the output y, given a set of features as the input x.

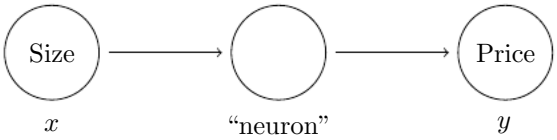


Figure 1.1.1: Simplest possible neural network where we are given the size of a home and predict its price.

Figure 1.1.1 represents the simplest possible neural network. The neuron is the part of the neural network that tries to learn a function that maps $x \to y$. Here, we only have one neuron, which is why it is the simplest case. We can make more complicated neural networks by stacking neurons. Consider the case where we not only have the size but the number of bedrooms, zip code, wealth as well. We can represent the extra inputs inside of a neural network as shown in Figure 1.1.2.

Neural networks work well because we only need to feed the algorithm supervised data, without specifying what all the intermediate values may be, as we did in Figure 1.1.2 with family size, walkability, and schooling. Instead of connecting together only a few inputs, we can connect all of the inputs together. Layers with all inputs connected to the output are known as fully-connected layers. The general structure of a neural network will fully connected layers can be seen in Figure 1.1.3.

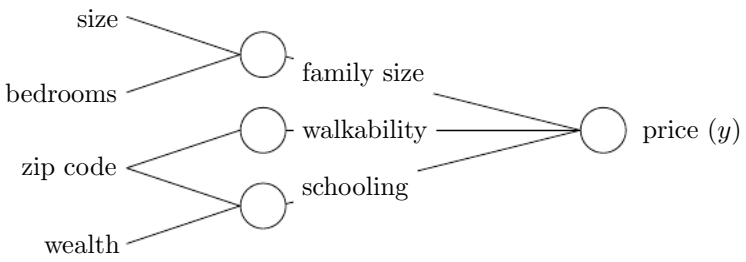


Figure 1.1.2: In a neural network with more neurons, the intermediate connections between inputs may represent their own values.

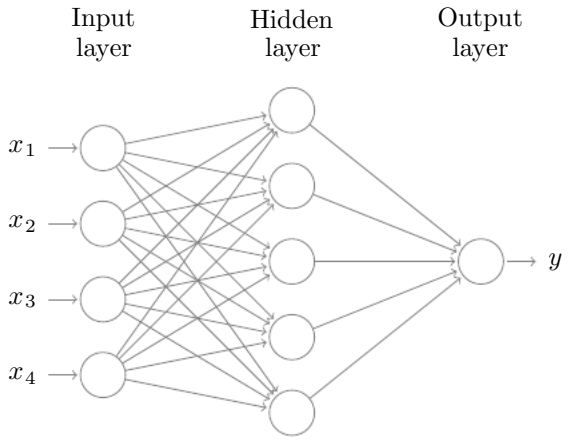


Figure 1.1.3: Neural network with fully connected layers

1.1.2 Supervised learning with neural networks

Almost all of the hype around machine learning has been centered around supervised learning. A supervised learning algorithm takes in a set of features and outputs a number or a set of numbers. Some examples can be seen in Table 1.1.

Input (x)	Output (y)	Application
Home features	Price	Real estate
Ad, user info	Click on ad? (0/1)	Online advertising
Image	Object (1,..., 1000)	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, radar info	Position of other cars	Autonomous driving

Table 1.1: Examples of supervised learning

As we have already seen, we can change the connections between layers to form different types of neural networks. Indeed, changing the structure of layers has led to many of the breakthroughs in deep learning and we have formulated tasks in which it would be beneficial not using a neural network with fully connected layers. For example, while real estate and online advertising we may use a neural network with fully-connected layers, photo tagging may use convolutional neural networks (CNNs), sequenced data may use recurrent neural networks (RNNs), and for something more complicated like autonomous driving, we may use some custom hybrid neural network. Figure 1.1.4 shows the basic structure of different neural networks, although we will go more in-depth into how each works in this text.

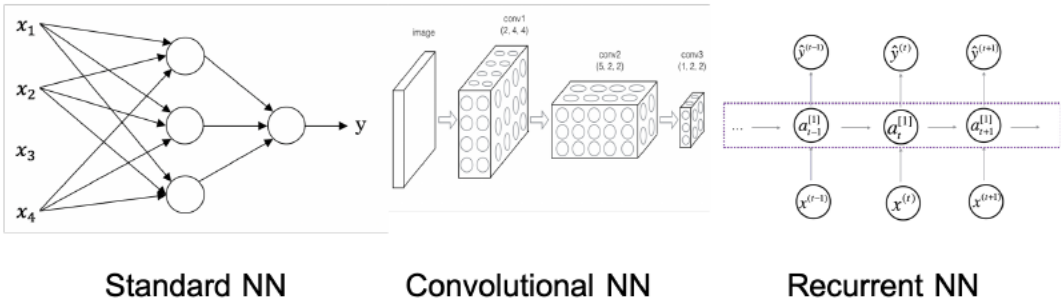


Figure 1.1.4: Different types of neural network architectures

Supervised learning can be both structured and unstructured. Structured data may have come in the form of database entries, while unstructured data may be audio, image clips, or text. Historically, unstructured data like image recognition has often been a harder problem for computers to solve, while humans have little difficulty solving these types of problems. Neural networks have given computers a lot of help when interpreting unstructured data.

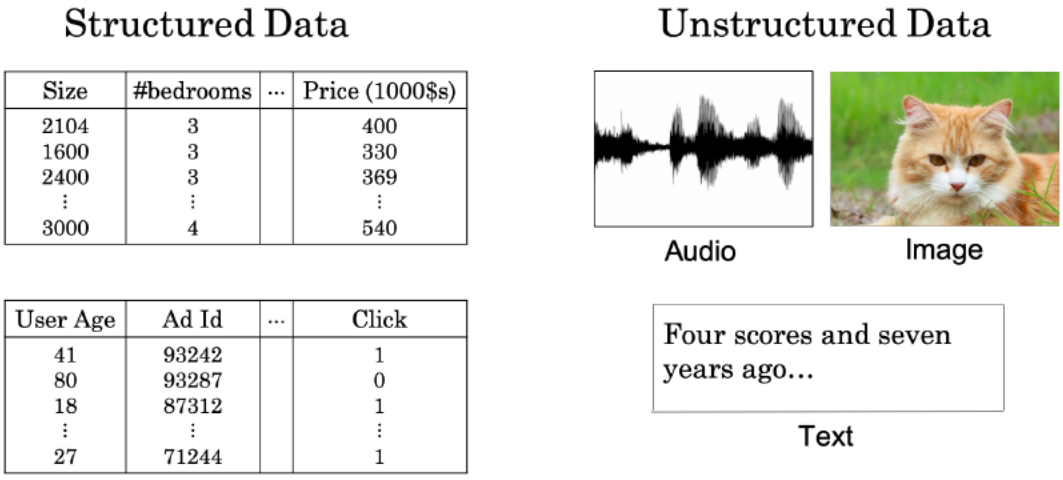


Figure 1.1.5: Structured vs unstructured data

1.2 Logistic regression as a neural network

1.2.1 Notation

Each training set will be composed of m training examples. We may denote m_{train} to be the number of training examples in the training set, and m_{test} to be the number of examples in the testing set. The ith input instance will be defined as $(x^{(i)}, y^{(i)})$, where $x^{(i)} \in \mathbb{R}^{n_x}$ and each $y^{(i)} \in \{0, 1\}$. To make notation more concise, we will use X to denote the matrix formed by

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix}, \tag{1.2.1}$$

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

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}, \tag{1.2.2}$$

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

1.2.2 Binary classification

A binary classifier takes some input and classifies that input in either two categories, typically yes (1) or no (0). For example, an image of a cat may be classified as a cat or non-cat. We will use y to denote the output as either a 0 or 1.

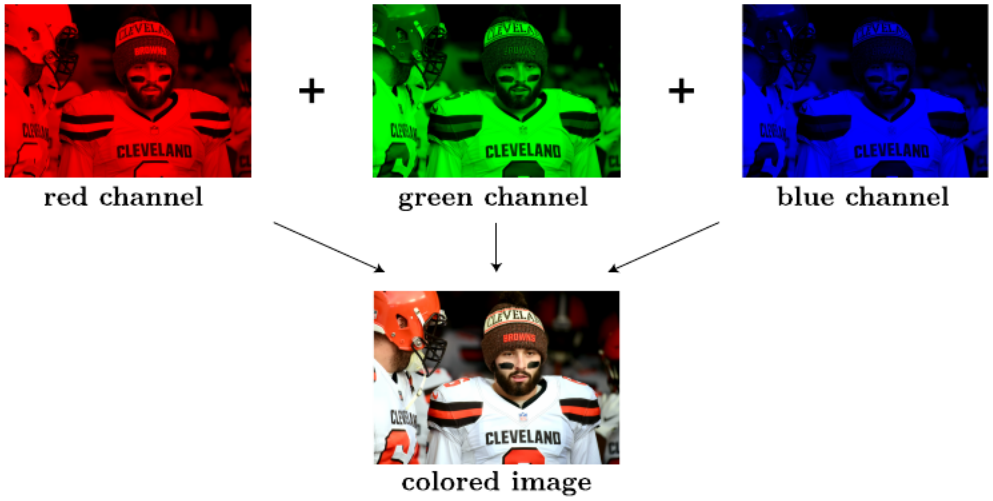


Figure 1.2.1: Colored images can be represented with the addition of red, green, and blue channels.

Each color can be represented using the colors red, green, and blue. Each pixel on a colored image can be represented as a tuple, storing the amount of red, green, and blue inside of each pixel. For images, we say that the red, green, and blue channel corresponds the amount of that color in each pixel as shown in Figure 1.2.1. For an image of size $n \times m$, we will define our input as a single column vector x that stacks the red values on the blue values on the green values as shown

$$x = \begin{bmatrix} \text{red values} \\ \text{blue values} \\ \text{green values} \end{bmatrix}. \tag{1.2.3}$$

We will use n_x to denote the input size of our vector. In this case, our input size is

$$n_x = 3nm, \tag{1.2.4}$$

since each channel is size $(n \times m)$ and there are 3 channels.

1.2.3 Logistic Regression

Given an input feature $x \in \mathbb{R}^{n_x}$, such as the pixels of an image, we would like to find an algorithm that makes a prediction $\hat{y}$, where

$$\hat{y} = P(y = 1 \mid x). \tag{1.2.5}$$

Inside of our algorithm we will define weights $w \in \mathbb{R}^{n_x}$ and a bias term $b \in \mathbb{R}$, where our output is

$$\hat{y} = \sigma\left(w^T x + b\right). \tag{1.2.6}$$

The sigmoid function σ is a function is used to transform the input between $0 \leq \hat{y} \leq 1$, where

$$\sigma = \frac{1}{1 + e^{-z}}. \tag{1.2.7}$$

We will denote the inner part of the sigmoid function as

$$z = w^T x + b. \tag{1.2.8}$$

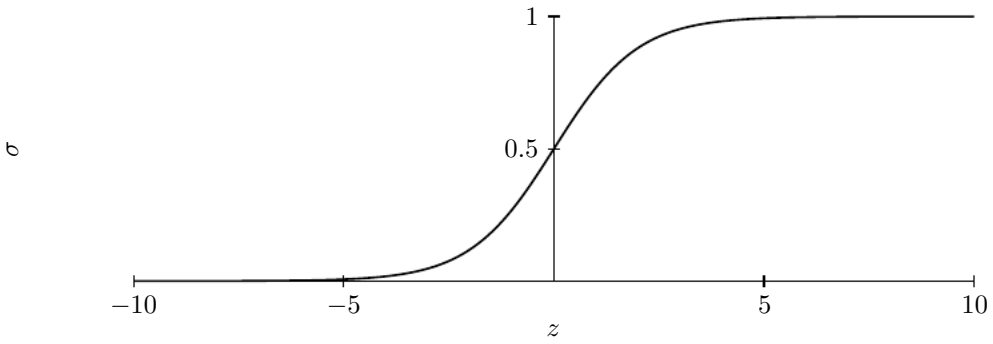


Figure 1.2.2: Sigmoid function

1.2.4 Logistic regression cost function

Logistic regression is be a supervised learning algorithm where we train on the training examples $\left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}$ and would like $\hat{y}^{(i)} \approx y^{(i)}$. While many machine learning algorithms use a squared error cost function, logistic regression does not work well when using squared error because many local optima arise. Instead, we will use the loss function

$$\mathcal{L}\left(\hat{y}, y\right) = -\left(y \log \hat{y} + (1-y) \log\left(1-\hat{y}\right)\right). \tag{1.2.9}$$

Intuitively, consider the following two scenarios. When $y = 1$, we will end up with

$$\mathcal{L}\left(\hat{y}, 1\right) = -\log \hat{y}, \tag{1.2.10}$$

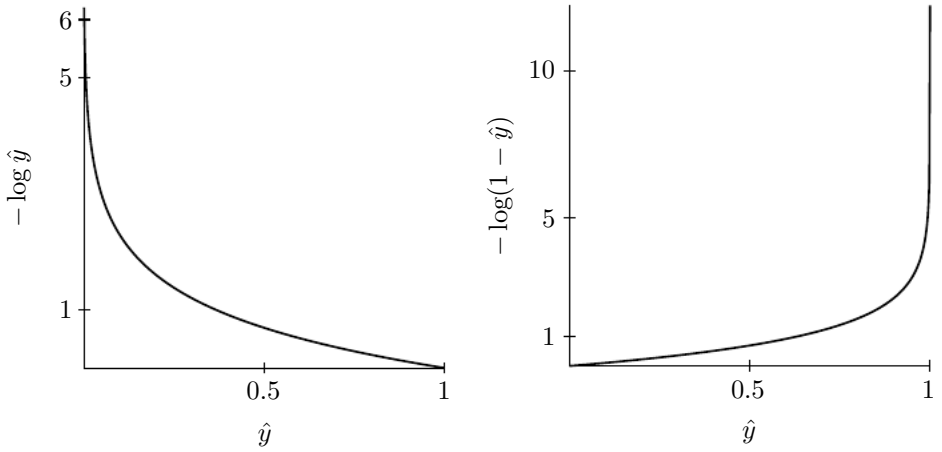
which is shown in Figure 1.2.3a. The second scenario occurs when $y = 0$. In that case, we will have

$$\mathcal{L}(\hat{y}, 0) = -\log(1-\hat{y}), \tag{1.2.11}$$

which is shown in Figure 1.2.3b. When small $\hat{y}$ are inputted, we will have a small loss, and when $\hat{y}$ approaches 1, our estimates will be way off, producing a large cost.

The loss function is a measure of how well the classifier is doing on a single example. The cost function, defined as the average loss

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right). \tag{1.2.12}$$



(a) Loss function when $y = 1$

(b) Loss function when $y = 0$

Figure 1.2.3: A binary classifier's loss function based on if $y = 1$ or $y = 0$.

1.2.5 Gradient descent

For many logistic regression problems, our function will look bowl-shaped, which is known as a convex function shown in Figure 1.2.4. With convex functions, there is a clear minimum of the function and we would like to find the representations of w and b that will put us at the minimum.

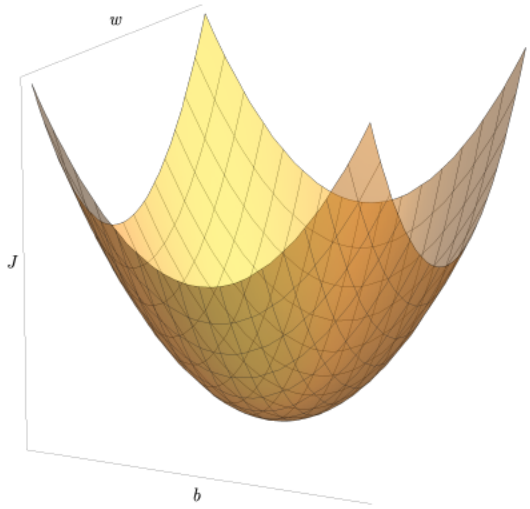


Figure 1.2.4: An example of what our cost function might look like with the inputs w and b.

For logistic regression, we generally start by initializing our weights to be zero. Gradient descent steps in the direction of steepest descent. If we start at some point that is not the minimum, we will move in the direction of the minimum, until we converge near the minimum. In one dimension, we can define the gradient descent algorithm as

$$w = w - \alpha \frac{\partial J(w,b)}{\partial w} \tag{1.2.13}$$

$$b = b - \alpha \frac{\partial J(w,b)}{\partial b} \tag{1.2.14}$$

where we will first calculate both of the partial derivative terms and then store it inside of the variables in the left hand side.

1.2.6 Computation graphs

Consider the function

$$y(a,b,c) = 3(a+bc). \tag{1.2.15}$$

To compute the output requires three steps, first compute bc, then add that result to a, and then multiply the added result by 3. We can represent this function as the computational graph shown in Figure 1.2.5. Each forward propagation in the computation graph produces a number that gets closer to the result.

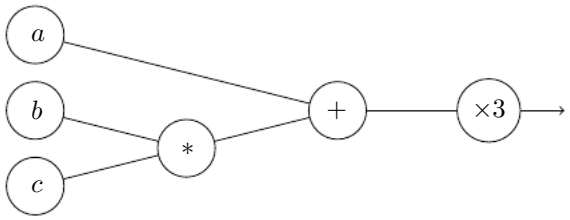


Figure 1.2.5: Computation graph for $3(a+bc)$

1.2.7 Derivatives with a computational graph

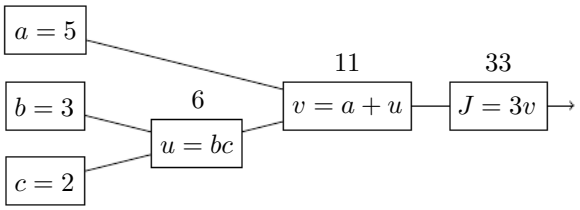


Figure 1.2.6: Computation graph for $3(a + bc)$

Consider the updated computational graph in Figure 1.2.6. Now, suppose that we want to find how J changes with respect to v, that is $\partial J/\partial v$ at $v = 11$. Since we know that $J = 3v$, we can take the derivative of the function giving us

$$\frac{\partial J}{\partial v} = 3, \tag{1.2.16}$$

which in this case is not based on the input at all. Now if we wish to find $\partial J/\partial a$, we can use the chain rule to break

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a}. \tag{1.2.17}$$

Since we already calculated the value of $\partial J/\partial v = 3$, we only need to find $\partial v/\partial a$. Since $v = a+u$, differentiating both sides with respect to a will give us

$$\frac{\partial v}{\partial a} = 1. \tag{1.2.18}$$

Now, using Equation 1.2.17

$$\frac{\partial J}{\partial a} = 3. \tag{1.2.19}$$

To find $\partial J/\partial u$, we would use a similar process how we found $\partial J/\partial a$ and we will again end up with $\partial J/\partial u = 3$. To find $\partial J/\partial b$, we need

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial u}{\partial b} \tag{1.2.20}$$

$$= 3 \cdot \frac{\partial u}{\partial b}. \tag{1.2.21}$$

Since $u = bc$, differentiating both sides with respect to b gives us $\partial u/\partial b = c$, where c in this case is 2, so

$$\frac{\partial J}{\partial b} = 6. \tag{1.2.22}$$

A similar process would be carried out to find $\partial J/\partial c = 9$.

1.2.8 Logistic regression gradient descent

With our logistic regression computational graph defined in Figure 1.2.7, we can again work backward calculating the changes in $\mathcal{L}$ with respect each of the variables. Differentiating both sides of our loss function with respect to a gives us

$$\frac{\partial \mathcal{L}(a, y)}{\partial a} = -\frac{y}{a} + \frac{1 - y}{1 - a}. \tag{1.2.23}$$

We also find that

$$\frac{\partial a}{\partial z} = a(1 - a), \tag{1.2.24}$$

so

$$\frac{\partial \mathcal{L}}{\partial z} = a - y. \tag{1.2.25}$$

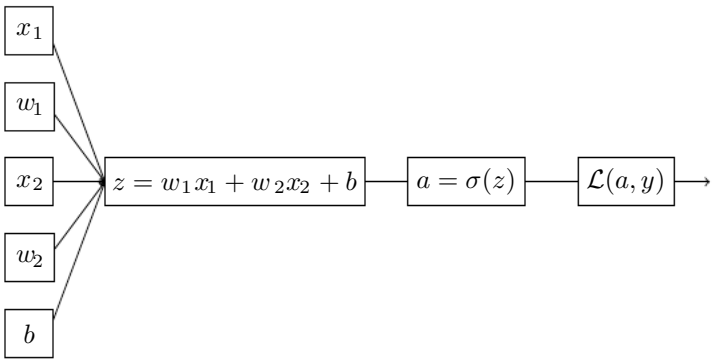


Figure 1.2.7: Computation graph for logistic regression

Now back propagating towards the inputs, we have

$$\mathcal{L}_{x_1} = (a - y) \cdot w_1 \tag{1.2.26}$$
$$\mathcal{L}_{x_2} = (a - y) \cdot w_2 \tag{1.2.27}$$
$$\mathcal{L}_{w_1} = (a - y) \cdot x_1 \tag{1.2.28}$$
$$\mathcal{L}_{w_2} = (a - y) \cdot x_2 \tag{1.2.29}$$
$$\mathcal{L}_b = (a - y). \tag{1.2.30}$$

In general, we can show

$$\mathcal{L}_{x_n} = (a - y) \cdot w_n \tag{1.2.31}$$
$$\mathcal{L}_{w_n} = (a - y) \cdot x_n. \tag{1.2.32}$$

1.2.9 Gradient descent on m examples

Recall that when we have m input instances to train on

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}), \tag{1.2.33}$$

where $a^{(i)} = \hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$. If we differentiate J with respect to w_1, we will have

$$\frac{\partial J(w, b)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}(a^{(i)}, y^{(i)})}{\partial w_1}. \tag{1.2.34}$$

In Equation 1.2.28, we calculated $\mathcal{L}_{w_1}$ to be $(a - y) \cdot x_1$, so

$$\frac{\partial J(w, b)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \cdot x_1 \,. \tag{1.2.35}$$

1.3 Python and vectorization

1.3.1 Vectorization

The aim of vectorization is to remove explicit for-loops in code. With deep learning datasets can be massive and it is vital to understand how vectorization works in order to improve algorithmic efficiency. Recall that we set $z = w^T x + b$. One implementation on how to calculate this would be

$$z = \sum_{i=1}^{n_x} w^T x^{(i)} + b, \tag{1.3.1}$$

which would end up being very slow. In contrast, we can use CPython to speed up the calculation, where the NumPy command will be

$$z = \text{np.dot}(w, x) + b. \tag{1.3.2}$$

Using built-in CPython functions enables parallelism much better than writing out explicit for-loops. When using neural networks, whenever possible, avoid explicit for-loops. A few more examples of calculations that can be vectorized include the matrix multiplication between Ax, which is also written as

$$Ax = \text{np.dot}(A, x). \tag{1.3.3}$$

To apply some operation, such as exponentiating, to each item in a vector v, write

$$v = \text{np.exp}(v). \tag{1.3.4}$$

1.3.2 Vectorizing logistic regression

For logistic regression, we need to find $a^{(i)} = \sigma(z^{(i)})$, m times, where $z^{(i)} = w^T x^{(i)} + b$. Instead of running explicit for-loops for each instance, we would like a way to vectorize the process of finding every value of a at once. Recall that we set

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)}, \end{bmatrix} \tag{1.3.5}$$

where $X \in \mathbb{R}^{n_x \times m}$. We also know that there is a weight associated with each input instance, so $w^T \in \mathbb{R}^{1 \times n_x}$. Carrying out the product between

$$w^T X = \begin{bmatrix} w^T x^{(1)} & w^T x^{(2)} & \dots & w^T x^{(m)} \end{bmatrix}. \tag{1.3.6}$$

Now, we can see that $b \in \mathbb{R}^{1 \times m}$, so taking the sum of $w^T X$ and b gives us

$$w^T X + b = \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & \dots & w^T x^{(m)} + b \end{bmatrix}. \tag{1.3.7}$$

In order to implement this in CPython, call

$$Z = \text{np.dot}(w.T,\ x) + b, \tag{1.3.8}$$

where $b \in \mathbb{R}$ and Python will expand it to be a $1 \times m$ matrix. The concept of expanding a number into a matrix in python is known as **broadcasting**.

Similar to how X stores the instances for $x^{(i)}$, we will define the matrix

$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix}. \tag{1.3.9}$$

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

$$A = \frac{1}{1 + \exp(-Z)}. \tag{1.3.10}$$

1.3.3 Vectorizing logistic regression's gradient computation

Recall from earlier that the change in b, which we will denote as db was equal to dz. We then wanted to update db by the average of all of the *nudges* over the training examples. Instead of initializing $db = 0$, looping over and adding every instance of $dz^{(i)}$, and dividing the result by m, we could instead call

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)} \tag{1.3.11}$$

$$= \frac{\text{np.sum}(dZ)}{m}, \tag{1.3.12}$$

where dZ is a row matrix formed by each $dz^{(i)}$ element. The update the change in weights, recall $dw = {}^1\!/_m \cdot \sum_{i=1}^{m} dx^{(i)} dz^{(i)}$, which can be expressed in vectorized for as

$$dw = \frac{\text{np.dot}(X, dZ)}{m}. \tag{1.3.13}$$

1.3.4 Broadcasting example

Consider the matrix

$$A = \begin{bmatrix} 56 & 0 & 4 & 68 \\ 1 & 102 & 52 & 8 \\ 2 & 135 & 99 & 1 \end{bmatrix} \tag{1.3.14}$$

and we want to find the percentage each element contributes to the sum of its column. For example, $A_{11} = 56/(56+1+2)$. To calculate the sum of each row, we would type

$$s = A.\text{sum}(\text{axis} = 0). \tag{1.3.15}$$

Axis 0 refers to the vertical columns inside of a matrix, while axis 1 refers to the horizontal rows in a matrix. Calling the method in Equation 1.3.15 will produce a vector of 4 elements, however, we would like to convert from $\mathbb{R}^4 \to \mathbb{R}^{1\times 4}$. To make this conversion, call

$$s = s.\text{reshape}(1, 4). \tag{1.3.16}$$

Now we can perform A/s in order to get the percentage each element contributes to the sum of the column. Dividing the matrix $A \in \mathbb{R}^{3\times 4}$ by the row vector $s \in \mathbb{R}^{1\times 4}$ will divide each of the three elements in a column by the value in the corresponding column of s. Since their sizes are different, this is known as **broadcasting**. A few other examples of broadcasting include adding a vector and real number

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}, \tag{1.3.17}$$

and adding a matrix in $\mathbb{R}^{m\times n}$ with a row vector in $\mathbb{R}^{1\times n}$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}. \tag{1.3.18}$$

Notice, when applying a command between a matrix in $\mathbb{R}^{m\times n}$ and a vector in $\mathbb{R}^{1\times n}$, the vector will expand to a matrix in $\mathbb{R}^{m\times n}$, where each row has the same elements. similarly, when combining a matrix in $\mathbb{R}^{m\times n}$ with a column vector in $\mathbb{R}^{m\times 1}$, the vector will expand to be in $\mathbb{R}^{m\times n}$, where each column is identical. An example of a column vector expanding is

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}. \tag{1.3.19}$$

1.3.5 A note on NumPy vectors

Consider the example

$$a = \text{np.random.randn}(5). \tag{1.3.20}$$

Calling a.shape will return the shape as (5,), which is considered a **rank one array**, which is neither a row vector nor a column vector. If we call $a.T$, nothing will be changed to the array, and it will still be in the shape (5,). Instead of using a rank one array, it is recommended to use natural matrices when dealing with neural networks. If we wanted a matrix of size $\mathbb{R}^{5\times 1}$ instead of the rank one array in Equation 1.3.20, we would call

$$a = \text{np.random.randn}(5, 1). \tag{1.3.21}$$

One good tip when dealing with a matrix of an unknown size is to first assert that it is a size we want, by calling

$$\text{assert(a.shape} == (5, 1)). \tag{1.3.22}$$

Chapter 2

Neural networks

2.1 Deep learning intuition

A model for deep learning is based on architecture and parameters. The architecture is the algorithmic design we choose, like logistic regression, linear regression, or shallow neural networks. The parameters are the weights and bias the model that takes in an instance as input and attempts to classify it correctly based on the parameters.

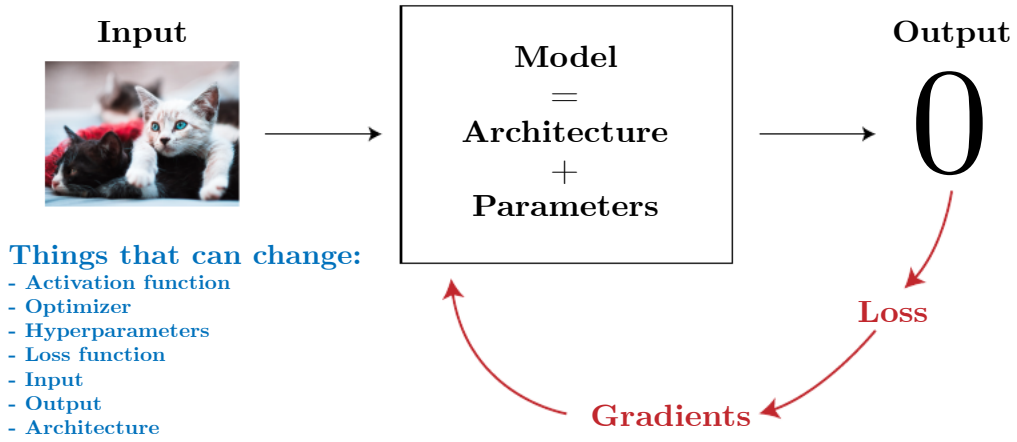


Figure 2.1.1: Where the model fits in and what can be adjusted

Consider if we wanted to use a multi-class classifier that not only predicts if an image is a cat or not a cat, but instead predicts from the image whether the image is a cat, dog, or giraffe. Recall that when using a binary classifier our weights were

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix}, \tag{2.1.1}$$

and our labels were single numbers, $y \in \{0, 1\}$. Now, let us add two more columns to the weights that will be the weights for the dog classifier and the giraffe classifier

$$w = \begin{bmatrix} \vdots & \vdots & \vdots \\ \text{cats} & \text{dogs} & \text{giraffes} \\ \vdots & \vdots & \vdots \end{bmatrix}. \tag{2.1.2}$$

Our weights are in $\mathbb{R}^{n_x \times 3}$.

To update our labels, we have a few options. First, let us put $y \in \{0, 1, 2\}$, where 0 corresponds to a cat image, 1 corresponds to a dog image, and 2 corresponds to a giraffe image. The second option is **one-hot encoding**, where we will have an array with each index mapping to an animal. With one-hot encoding, only one item in each labeled array will have a value of 1 (picture is of an animal), while the rest are 0 (picture is not of an animal). For example, if we have an image of a cat, our label would be

$$\begin{matrix} \text{cat} & \text{dog} & \text{giraffe} \\ [\,1 & 0 & 0\,]. \end{matrix} \tag{2.1.3}$$

While both of these options would work well in most cases, if we have a picture of both a cat and dog in the same image, our classifier will not work. Instead, we will use **multi-hot encoding**, which works similar to one-hot encoding, with the difference being multiple values can take on the value 1. So now if we have a picture of both a cat and dog at the same time, we can encode our label as

$$\begin{matrix} \text{cat} & \text{dog} & \text{giraffe} \\ [\,1 & 1 & 0\,]. \end{matrix} \tag{2.1.4}$$

The jth neuron in the ith layer will be labeled as $a_j^{[i]}$ as shown in Figure 2.1.2.

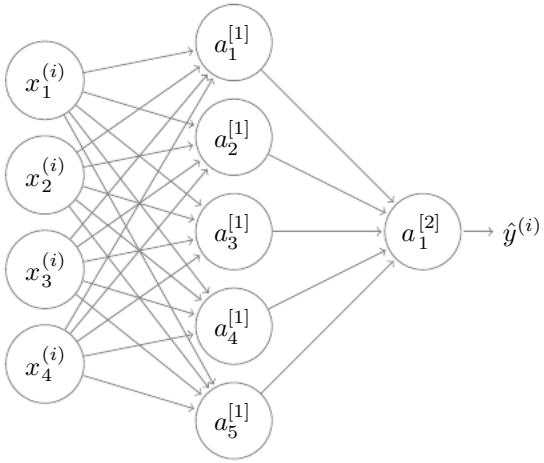


Figure 2.1.2: Notation for the course

As the layers increase, the activation neurons start to look at more complex items. For example, if we are working with a face classifier, the first layer might only detect edges, the second layer might put some of the edges together to find ears and eyes, and the third layer might detect the entire face. The process of extracting data from the neurons is known as **encoding**.

2.1.1 Day'n'night classification

Our goal is to create an image classifier that predicts from a photo taken outside if it was taken "during the day" (0) or "during the night" (1). From the cat example, we needed

about $10,000$ images to create a good classifier, so we estimate this problem is about as difficult and we will need about $10,000$ images to classify everything correctly. We will split the data up into both a training and testing set. In this case, about 80% of the data will be in the training set. In cases with more training data, we would give less of a percentage to the testing set and more of a percentage to the training set. We also need to make sure that the data is not split randomly, rather split proportionally to the group the data is in (80% of day examples and 80% of night examples go in the training set).

For our input, we would like to work with images that have the lowest possible quality, while still achieving good results. One clever way to choose the lowest possible quality images is to find out the lowest possible quality that humans can perfectly identify and then use that quality. After comparing to human performance, we find that $64 \times 64 \times 3$ pixels will work well as our image size.

Our output will be 0 for the day and 1 for the night. We will also use the sigmoid function as our last activation function because it maps a number in $\mathbb{R}$ to be between 0 and 1. A shallow neural network, one with only one hidden layer, should work pretty well with this example because it is a fairly straightforward task. For our loss function, we will use the **logistical loss**

$$\mathcal{L}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}). \tag{2.1.5}$$

2.1.2 Face verification

Our goal is to find a way for a school to use face verification for validating student IDs in facilities such as dining halls, gyms, and pools. The data we will need is a picture of every student labeled with their name. The input will be the data from the cameras as well as who they are attempting to be. In general, faces have more detail than the night sky, so we will need the input to be a larger size. Using the human test we used last time, we find a solid resolution to be $412 \times 412 \times 3$ pixels. We will make the output 1 if it's the correct person at the camera and 0 if it is not.

To compare the image we have of a person in our database with the image we have from a camera, we want to find a function to call on both of the images in order to determine if the person in the image is the same. We could directly compare the differences in pixels, however, that presents severe problems if the background lighting is different, the person grows facial hair, and the ID photo is outdated. Instead, we will use a deep network for our function as illustrated in Figure 2.1.3.

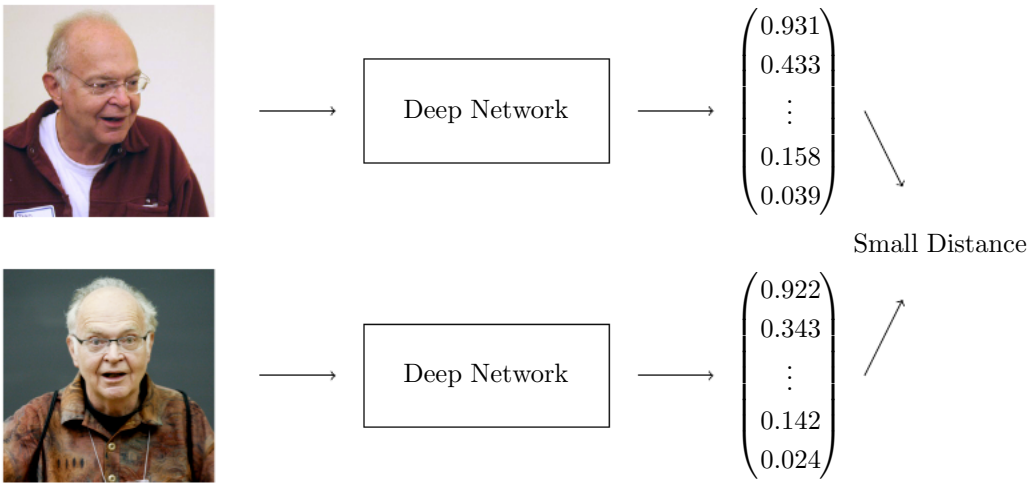


Figure 2.1.3: The architecture we will use to classify images will be deep networks.

We will set the distance threshold to be some number, and any distance less than that number we will classify with $y = 1$. Because the school will not have tons of data on the pictures of students, we will use a public face dataset, where we want to find images of the

same people to ensure that their encoding is similar and images of different people to ensure their encoding is different. We train our network, we will feed it **triplets**. Each instance will have an anchor (actual person), a positive example (the same person, minimize encoding distance), and a negative example (different person, maximize encoding distance).

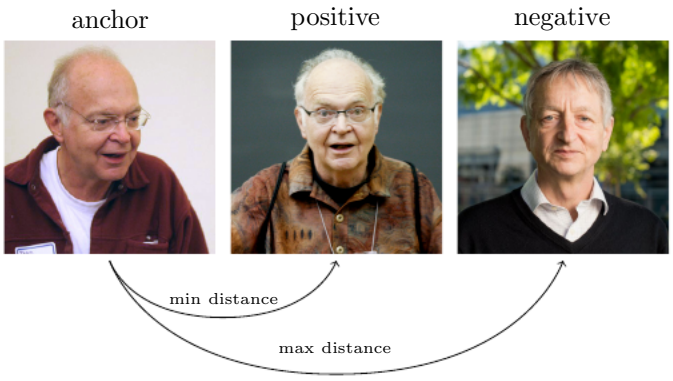


Figure 2.1.4: Ordered triplet input example

For our loss function, we can use

$$\mathcal{L} = ||Enc(A) - Enc(P)|| - ||Enc(A) - Enc(N)|| + \alpha, \tag{2.1.6}$$

where $||$ represents the L2 norm and Enc represents the encoding. The loss function presented makes sense because we will want $||Enc(A) - Enc(P)||$ to be minimized and $||Enc(A) - Enc(N)||$ to be maximized. To minimize a maximized function, use the negative of that function, which gives us our result of $-||Enc(A) - Enc(N)||$. The α term, called the **margin**, is used as a kickstart to our encoding function, to avoid the case where every weight in the deep network is zero, and we end up with a perfect loss. It is common to keep loss functions positive, so we generally train the maximum of the loss function and 0.

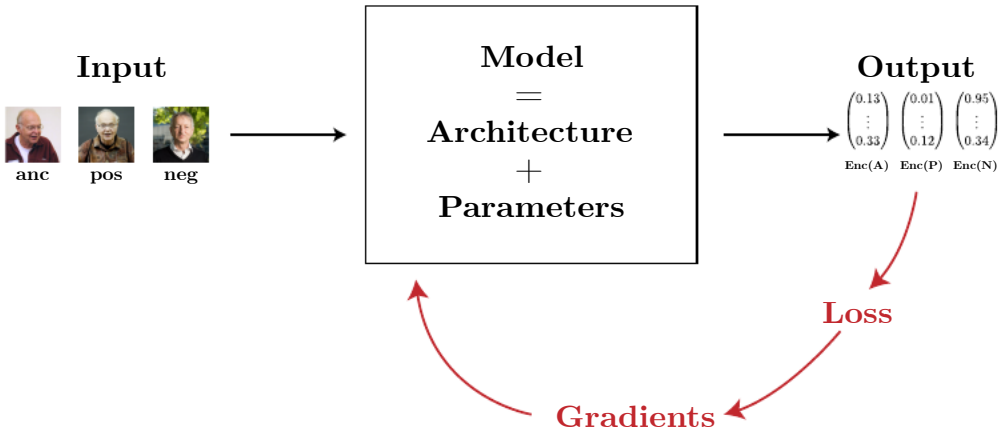


Figure 2.1.5: Updated model for face verification

2.1.3 Face recognition

In our school, we want to use face identification to recognize students in facilities. Now, instead of just verifying who the person is given their ID, we need to identify the person out of many people. We will use a lot of similar details from the face verification example, but instead of inputting 3 images and outputting 3 encodings, we will input one image and the encoding on that image. Then, we will compare the encoding our database, which contains encodings for every student and use a k-nearest neighbors algorithm to make comparisons. To go over the entire database, the runtime complexity will be $O(n)$.

For another example, suppose we have thousands of pictures on our phone of 20 different people and we want to make groups of the same people in folders. Using the encoding on each image, we could run a k-means clustering algorithm to find groups within the vector space that correspond to the same people.

2.1.4 Art generation (neural style transfer)

Given a picture, our goal is to make it *beautiful*. Suppose we have any data that we want. Our input will contain both content and the style we consider beautiful. Then, once we input a new content image, we would like to generate a style image based on the content.

(a) Content image **(b)** Style image

Figure 2.1.6: Input for art generation

The architecture we will use is first based on a model that understands images very well. The existing ImageNet models, for instance, are models that are perfectly well to use for an image understanding task. After the image is forward propagated through a few layers of networks, we will get information about the content in the image by looking at the neurons. We will call the information about the image $Content_C$. Then, giving the style matrix to the deep network, we will use the **grain matrix** to extract the $Style_S$ from the style image. More about the grain matrix will be discussed later in the course.

The loss function we can define as

$$\mathcal{L} = ||Content_C - Content_G|| + ||Style_S - Style_G||, \tag{2.1.7}$$

where $_G$ denotes the generated image.

2.1.5 Trigger word detection

When given a 10-second audio speech, we want to detect when the word "active" has been said. To create a model, we will need a lot of 10-second audio clips with as much of a variety in the voice as possible. We can break the input down into three segments: when "active" is being said, when a word that is not "active" is being said, and when nothing is being said. The sample rate of the input would be similar to how we found the resolution of images, determining the minimum viable amount that humans have no trouble with.

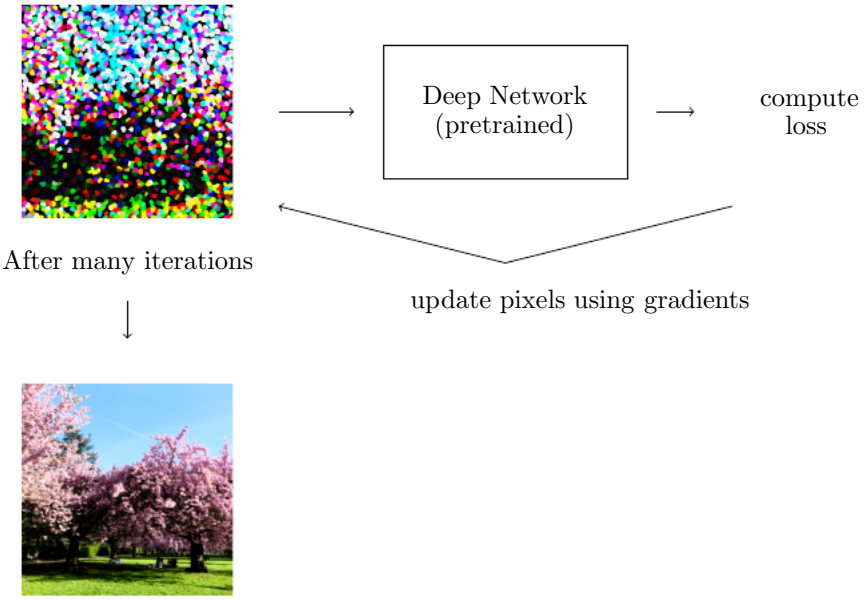


Figure 2.1.7: Art generation architecture

Figure 2.1.8: Input for the trigger word detection model. Green represents the time frame "active" is being said, red represents the time frame that non-"active" words are being said, and black represents the time frame nothing is being said.

The output will be a set of 0s and then a 1 after the word active is said. This output is generally easier to debug and works better for continuous models. The last activation we will use is a sigmoid and the architecture we will use should be recurrent neural networks. We can generate data for our methods by collecting positive words, negative words, and background noise and then adding different combinations of the data together to form 10-second clips.

2.2 Shallow neural networks

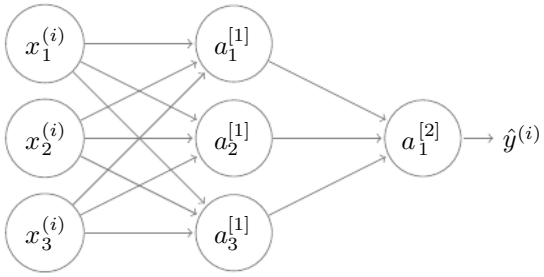


Figure 2.2.1: Shallow neural network

2.2.1 Neural networks overview

In each layer of a neural network, we will calculate both the linear activation, and then map the linear activation to be between 0 and 1. To calculate the linear activation in the ith layer

$$z^{[i]} = W^{[i]}x + b^{[i]}. \tag{2.2.1}$$

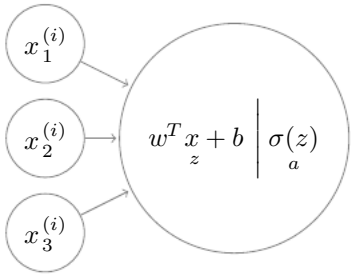


Figure 2.2.2: Each neuron in an activation layer represents two computations: the linear weights computation plus the bias and then squeezing that number between 0 and 1.

To calculate the non-activation the ith layer, we would use the sigmoid function

$$a^{[i]} = \sigma(z^{[i]}). \tag{2.2.2}$$

Only once we get to the final layer would we calculate the loss.

2.2.2 Neural network representations

The hidden layer refers to data that is not seen in the training set. To denote the input layer x, we may also use the notation $a^{[0]}$. When we count neural network layers, we do not include the input layer in that count, however, we do include the output layer. We will also refer to the nodes in the hidden layers and output layer as neurons.

2.2.3 Computing a neural network's output

In the shallow neural network pictured in Figure 2.2.5, for the hidden layer, we must calculate

$$\begin{cases} z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]} \\ a_1^{[1]} = \sigma(z_1^{[1]}) \\ \\ z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]} \\ a_2^{[1]} = \sigma(z_2^{[1]}) \\ \\ z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]} \\ a_3^{[1]} = \sigma(z_3^{[1]}) \end{cases} . \tag{2.2.3}$$

Implementing the assignments in Equation 2.2.3 would require an inefficient for-loop. Instead, we will vectorize the process as follows

$$z^{[1]} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x_1 + b_1^{[1]} \\ w_2^{[1]T} x_2 + b_2^{[1]} \\ w_3^{[1]T} x_3 + b_3^{[1]} \end{bmatrix} . \tag{2.2.4}$$

The dimensions of the weights matrix will be the number of weights in the hidden layer by the number of inputs. The bias vector will be a column vector with the same number of rows as the weight matrix. We will call the weight matrix $W^{[1]}$ and the bias vector $b^{[1]}$. For our activation, it follows that

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix} = \sigma(z^{[1]}). \tag{2.2.5}$$

Putting both the values of z and a together, we now only have to calculate

$$\begin{cases} z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[1]} = \sigma(z^{[1]}) \end{cases}. \tag{2.2.6}$$

Instead of writing x, we will write in its place $a^{[0]}$, which gives us the assignments

$$\begin{cases} z^{[1]} = W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} = \sigma(z^{[1]}) \end{cases}. \tag{2.2.7}$$

We prefer the notation $a^{[0]}$ in this case, because it will make deeper layers more natural to reference. Now for the second layer, we can use a similar process to show

$$\begin{cases} z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} = \sigma(z^{[2]}) \end{cases}. \tag{2.2.8}$$

From Figure 2.2.5, we see that there is only 1 neuron in activation layer 2 with 3 inputs. From that data, it implies that the dimensions of the weight matrix are 1×3 multiplied by the 3×1 input matrix and added to the 1×1 bias term.

2.2.4 Vectorizing across multiple examples

Previously, we have only calculated the values from one a single input x. For m training examples, we would be required to run a loop over every single example in order to calculate each corresponding $a^{[2](i)}$ value, where the index i is the ith training example. The corresponding for-loop can be shown in Algorithm 1.

Algorithm 1 Forward propagation

for $i \in \{1, \ldots, m\}$ **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

Earlier, we defined the matrix X to store all of the input instances as

$$X = \begin{bmatrix} | & & | \\ x^{(1)} & \ldots & x^{(m)} \\ | & & | \end{bmatrix}, \tag{2.2.9}$$

where $X \in \mathbb{R}^{n_x \times m}$. If we instead define

$$Z^{[1]} = W^{[1]}X + b^{[1]}, \tag{2.2.10}$$

where we broadcast b as a column vector to be a matrix with m columns. It follows that

$$A^{[1]} = \sigma(Z^{[1]}). \tag{2.2.11}$$

Now, in each of the columns in A and Z we will have

$$Z^{[1]} = \begin{bmatrix} | & & | \\ z^{[1](1)} & \ldots & z^{[1](m)} \\ | & & | \end{bmatrix}, \qquad A^{[1]} = \begin{bmatrix} | & & | \\ a^{[1](1)} & \ldots & a^{[1](m)} \\ | & & | \end{bmatrix}. \tag{2.2.12}$$

Horizontal nodes will correspond to different traning examples and vertical nodes correspond to different neurons in the hidden unit. The complete vectorization across multiple examples will be:

$$\begin{cases} Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]} \\ A^{[1]} = \sigma(Z^{[1]}) \\ Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(Z^{[2]}) \end{cases}. \tag{2.2.13}$$

2.2.5 Activation functions

Previously when forward propagating, we used σ as our activation function, although that is just one choice. Recall the graph of

$$\sigma = \frac{1}{1+e^{-z}} \tag{2.2.14}$$

in Figure 2.2.3 was used to map z from any number in $\mathbb{R}$ to be between 0 and 1.

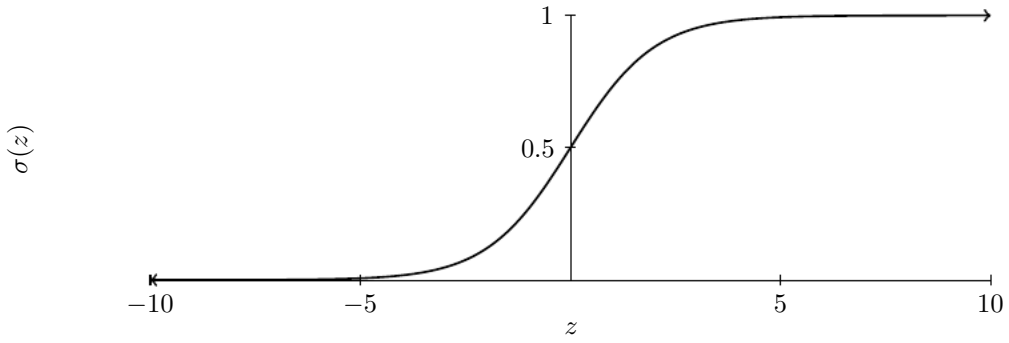


Figure 2.2.3: Sigmoid function

Let us denote our activation function as g, so that

$$a^{[i]} = g(z^{[i]}). \tag{2.2.15}$$

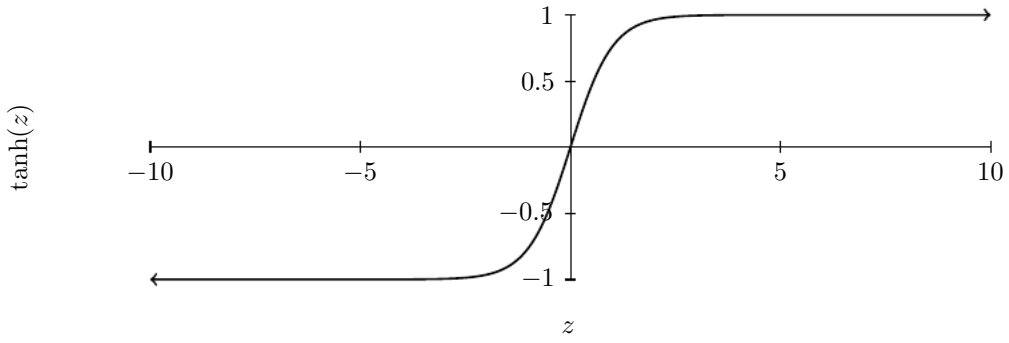


Figure 2.2.4: Hyperbolic tangent function as an activation function

An alternative to the traditional sigmoid function is the sigmoid function with a range from -1 to 1 that can be thought of as dragging the bottom of the sigmoid s from the line at 0 to the line at -1. After all of the calculations, we would end up finding that this function is the hyperbolic tangent function, so our a new activation function could be

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{2.2.16}$$

Generally, the hyperbolic tangent function tends to outperform the sigmoid function because it has the effect of centering the weights at 0. For the last layer, however, it is still common to use the sigmoid function to give a better interpretation of the data from a probabilistic perspective. Since different activation layers may have different activation functions, we will refer to the activation function in the ith layer as $g^{[i]}$.

As we approach larger numbers and smaller numbers for both the sigmoid and hyperbolic tangent functions, it is clear that the rate of change is decreasing drastically and is approaching no rate of change at the asymptotes. The **rectified linear unit (ReLU)** activation

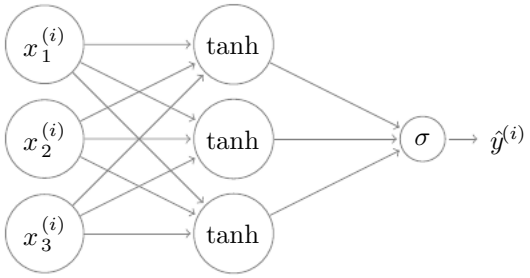


Figure 2.2.5: Different layers in the neural network may have different activation functions

function is often used to solve the problem at the infinities. The ReLU function is defined as

$$\text{ReLU}(z) = \max(0, z), \tag{2.2.17}$$

which gives the plot shown in Figure 2.2.6. The rate of change of the ReLU function will be 1 if $z > 0$ and 0 if $z < 0$. It is quite improbable that $z = 0.\bar{0}$, in which case our derivative is undefined and we should manually set the result to either 0 or 1 in that case.

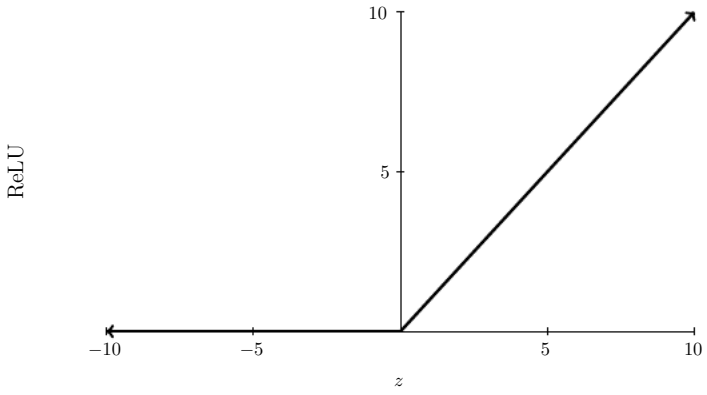


Figure 2.2.6: ReLU activation function

For binary classification problems, where $y \in \{0, 1\}$, it is common practice to use the sigmoid activation function in the last layer and the ReLU activation function everywhere else. We prefer the ReLU activation function in practice because it is often much easier and faster to handle derivatives.

The last activation function is the **leaky ReLU**, which behaves similar to the ReLU function with the exception being how negative numbers are handled. Instead of making all negative number 0, the leaky ReLU uses some scaling factor multiplied by z. The most common scaling factor is 0.01, which corresponds to the leaky ReLU function

$$\text{leaky ReLU}(z) = \max(0.01z, z). \tag{2.2.18}$$

In practice, it is often hard to determine which activation function should be used. It is quite common to test different combinations of activation functions in order to find which one works the best.

2.2.6 Why use non-linear activation functions?

Consider the shallow neural network pictured in Figure 2.2.8. If we used a linear activation function, that is

$$g^{[i]} = z^{[i]}, \tag{2.2.19}$$

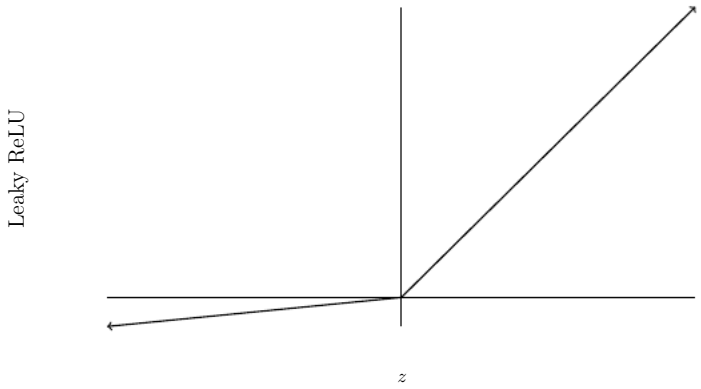


Figure 2.2.7: Leaky ReLU activation function

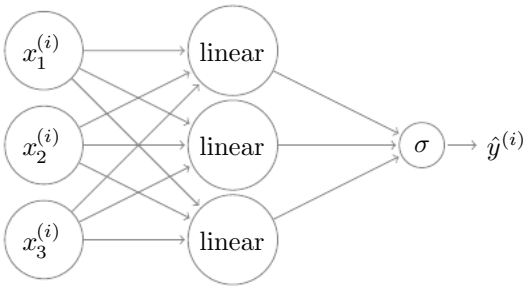


Figure 2.2.8: Different layers in the neural network may have different activation functions

then we have

$$\begin{cases} a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \end{cases} . \tag{2.2.20}$$

Substituting $W^{[1]}x + b^{[1]}$ in for $a^{[1]}$ in the second equation, we have

$$a^{[2]} = W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]}. \tag{2.2.21}$$

Expanding out the matrix, we have

$$(W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]}), \tag{2.2.22}$$

which is a linear equation of x. Since this is a linear hidden layer, storing deeper and deeper sets of linear weights on top of each other will never change our function from being linear, so more weights end up being useless. When using a linear activation function for preliminary calculations, the entire function tends to appear just like a logistic regression problem.

In the case of linear regression where $\hat{y} \in \mathbb{R}$, such as when housing prices are being predicted, we may use a linear function in the output layer of the neural network in order to map the set of the real number, but other than that case, linear activation functions should be avoided.

2.2.7 Derivatives of activation functions

With a sigmoid activation function

$$g(z) = \frac{1}{1+e^{-z}}, \tag{2.2.23}$$

we can write the derivative in terms of the original function using some clever algebraic manipulation that follows:

$$\frac{\partial \sigma}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} \tag{2.2.24}$$

$$= \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \tag{2.2.25}$$

$$= \frac{1+e^{-z}-1}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \tag{2.2.26}$$

$$= \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right) \cdot \frac{1}{1+e^{-z}} \tag{2.2.27}$$

$$= (1-\sigma) \cdot \sigma. \tag{2.2.28}$$

When using the hyperbolic tangent function as our activation function, we have

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{2.2.29}$$

The derivative of the hyperbolic tangent function can also be written in terms of its input

$$\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z). \tag{2.2.30}$$

For the ReLU function

$$g(z) = \max(0, z), \tag{2.2.31}$$

the derivative is

$$g'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}. \tag{2.2.32}$$

In there rare case where we have $g'(z=0)$, the derivative is techincally undefined, although in practice we choose a value of either 0 or 1. The choice tends to be very insignificant.

For the leaky ReLU function

$$g(z) = \max(0.01z, z), \tag{2.2.33}$$

the derivative is

$$g'(z) = \begin{cases} 1 & z > 0 \\ 0.01 & z < 0 \end{cases}. \tag{2.2.34}$$

We will treat the case where $z = 0$ for the leaky ReLU similar to the case where $z = 0$ for the ReLU, in which we choose either 0 or 1.

2.2.8 Gradient descent for neural networks

For the shallow neural network with one input unit, one hidden unit, and one output unit, we will have the parameters $W^{[1]T} \in \mathbb{R}^{n^{[1]} \times n^{[0]}}$, $b^{[1]} \in \mathbb{R}^{n^{[1]} \times 1}$, $W^{[2]T} \in \mathbb{R}^{n^{[2]} \times n^{[1]}}$, and $b^{[2]} \in \mathbb{R}^{n^{[2]} \times 1}$. So far we have only seen examples where $n^{[2]} = 1$. We are also denoting $n^{[0]}$ as n_x and $n^{[i]}$ to be the number of nodes in the ith layer.

For now, assume we are doing binary classification with the cost function

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y), \tag{2.2.35}$$

where $\hat{y} = a^{[2]}$ in this case and our loss function is

$$\mathcal{L}(\hat{y}, y) = -y \log \hat{y} - (1-y) \log(1-\hat{y}). \tag{2.2.36}$$

We then make updates to our parameters with the gradient descent algorithm shown in Algorithm 2.

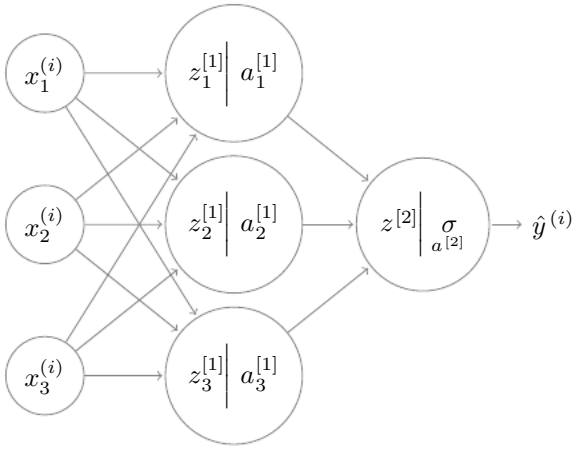


Figure 2.2.9: Each neuron can be broken into two parts, the linear activation z and then non-linear activation a

Algorithm 2 Gradient descent

repeat
Compute the predictions $\hat{y}^{(i)}$ for $i \in \{1, \ldots, m\}$
Compute the derivatives of the parameters $dW^{[\ell]}, db^{[\ell]}$ with respect J
Update the weights and bias with $W^{[\ell]} = W^{[\ell]} - \alpha\, dW^{[\ell]}$ and $b^{[\ell]} = b^{[\ell]} - \alpha\, db^{[\ell]}$
until the cost function J converges

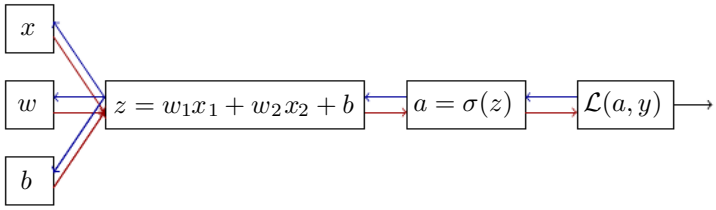


Figure 2.2.10: Computation graph for logistic regression

Recall that when we were using logistic regression, we had the computational graph in Figure 2.2.10, where the red arrows forward propagation and the blue arrows represent backward propagation. We previously calculated

$$\mathcal{L}_a = \frac{y-1}{a-1} - \frac{y}{a} \tag{2.2.37}$$

$$\mathcal{L}_z = \mathcal{L}_b = a - y \tag{2.2.38}$$

$$\mathcal{L}_x = w(a-y) \tag{2.2.39}$$

$$\mathcal{L}_w = x(a-y). \tag{2.2.40}$$

In the two layer computational graph pictured in Figure 2.2.11, it is clear that $a^{[2]}$, $z^{[2]}$, $W^{[2]}$, and $b^{[2]}$ will have similar rates of change to the loss function to a, z, w, and b when using logistic regression, so we have the derivatives

$$\mathcal{L}_{a^{[2]}} = \frac{y-1}{a^{[2]}-1} - \frac{y}{a^{[2]}} \tag{2.2.41}$$

$$\mathcal{L}_{z^{[2]}} = a^{[2]} - y \tag{2.2.42}$$

$$\mathcal{L}_{b^{[2]}} = a^{[2]} - y \tag{2.2.43}$$

$$\mathcal{L}_{W^{[2]}} = a^{[1]T} \cdot (a^{[2]} - y). \tag{2.2.44}$$

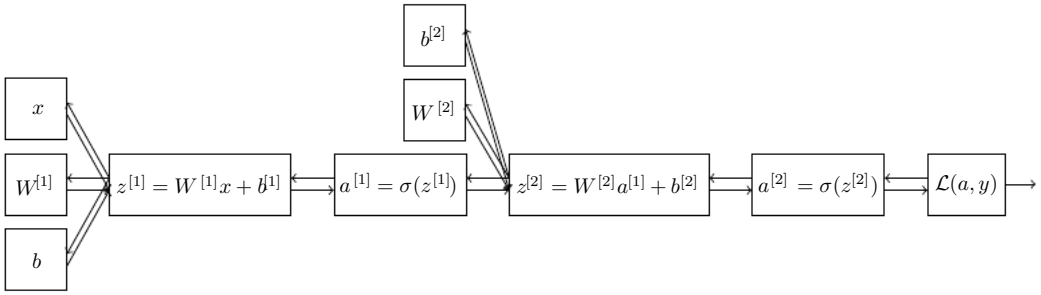


Figure 2.2.11: Two layer neural network computational graph

We can calculate $\mathcal{L}_{a^{[1]}}$ from the chain rule

$$\frac{\partial \mathcal{L}}{\partial a^{[1]}} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial a^{[1]}} \tag{2.2.45}$$

$$= (a^{[2]} - y) \cdot W^{[2]T}. \tag{2.2.46}$$

Going backward to $\mathcal{L}_{z^{[1]}}$, we can use the result from $\mathcal{L}_{a^{[1]}}$

$$\frac{\partial \mathcal{L}}{\partial z^{[1]}} = \frac{\partial \mathcal{L}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \tag{2.2.47}$$

$$= ((a^{[2]} - y) \cdot W^{[2]T}) \cdot (z^{[1]} \cdot (1 - z^{[1]})). \tag{2.2.48}$$

Now we can get back to the weight and bias terms in the first layer, where the rate of change with respect to $W^{[1]}$ is

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial W^{[1]}} \tag{2.2.49}$$

$$= W^{[2]T}(a^{[2]} - y) \cdot z^{[1]}(1 - z^{[1]}) \cdot xT, \tag{2.2.50}$$

and the rate of change with respect to b is

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}} \tag{2.2.51}$$

$$= W^{[2]T}(a^{[2]} - y) \cdot z^{[1]}(1 - z^{[1]}). \tag{2.2.52}$$

What we have written is for the case of one input example, however, we can extend to all the input examples by using vectorization, where

$$\mathcal{L}_{Z^{[2]}} = A^{[2]} - Y. \tag{2.2.53}$$

For $\mathcal{L}_{W^{[2]}}$, we will need to conisder the cost function inside of our rate of change, so we get

$$\mathcal{L}_{W^{[2]}} = \frac{1}{m} \mathcal{L}_{Z^{[2]}} A^{[1]T}. \tag{2.2.54}$$

We will write $\mathcal{L}_{b^{[2]}}$ using the CPython notation of

$$\mathcal{L}_{b^{[2]}} = \frac{1}{m} \cdot \text{np.sum}(\mathcal{L}_{Z^{[2]}}, \text{axis} = 1, \text{keepdims} = \text{True}). \tag{2.2.55}$$

We will have $\mathcal{L}_{Z^{[1]}} \in \mathbb{R}^{n^{[1]} \times m}$, where

$$\mathcal{L}_{Z^{[1]}} = W^{[2]T} \mathcal{L}_{Z^{[2]}} * g^{[1]\prime}(Z^{[1]}), \tag{2.2.56}$$

where $g^{[1]}$ is the activation function in layer 1, and $*$ is the element product operator. Finally the change to the weights and bias in layer 1 will be

$$\mathcal{L}_{W^{[1]}} = \frac{dZ^{[1]} X^T}{m} \tag{2.2.57}$$

and

$$\mathcal{L}_{b^{[1]}} = \frac{1}{m} \cdot \text{np.sum}(\mathcal{L}_{Z^{[1]}}, \text{axis} = 1, \text{keepdims} = \text{True}). \tag{2.2.58}$$

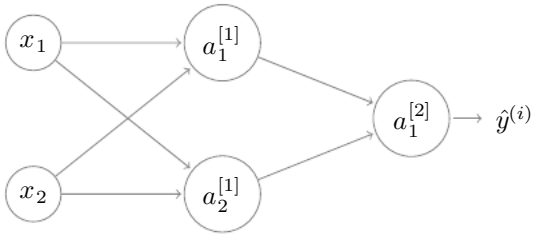


Figure 2.2.12: Two layer neural network

2.2.9 Random Initialization

Consider if we initialized

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \tag{2.2.59}$$

then $a_1^{[1]} = a_2^{[2]}$. 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. With the same changes after each iteration, the two neurons will act as if they are the same, which means having two neurons is pointless since it does not add additional information. We call this error the **symmetry breaking problem**.

The solution to the problem is to initialize small non-zero random weights, where we initialize

$$W^{[1]} = \text{np.random.randn}((2, 2)) * 0.01. \tag{2.2.60}$$

We use the 0.01 term to make sure that our input is very small, since our hyperbolic tangent function and sigmoid function have more meaninful values near 0 and our function converge faster.

It turns out that our bias term b does not have a symmetry problem, so it is alright if we initialize it to be the zero column vector

$$b^{[1]} = \text{np.zeros}((2, 1)). \tag{2.2.61}$$

We can initialize the rest of the weights and bias terms in a similar fashion.

2.3 Deep neural networks

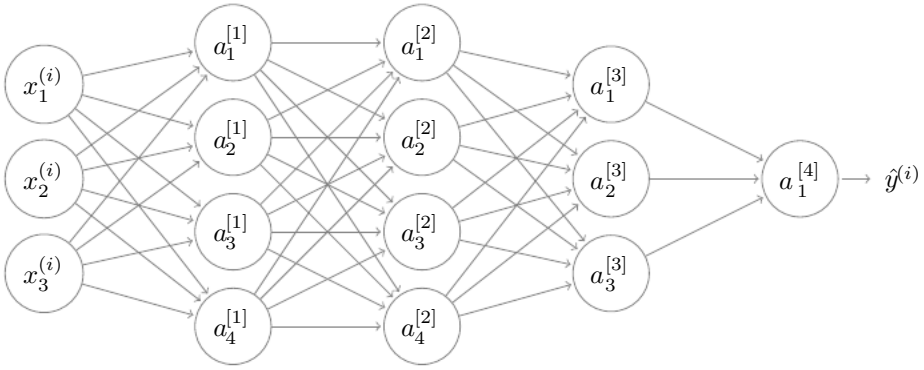


Figure 2.3.1: 4 layer neural network

For neural networks, we will use L to denote the number of layers and $n^{[\ell]}$ to denote the number of nodes in the nth layer. For example, in Figure 2.3.1, we have $L = 4$ and $n^{[1]}, n^{[2]} = 4$ and $n^{[3]} = 3$. To refer to the activations in each layer, we will use $a^{[\ell]}$ and to refer to the weights in each layer we will use $w^{[\ell]}$. Similarly, the linear activation and bias in each layer will be $z^{[\ell]}$ and $b^{[\ell]}$, respectively. We can refer to the input as $\ell = 0$ and the output as $\ell = L$.

2.3.1 Forward propagation in a deep network

To propagate a training example forward, using the network in Figure 2.3.1, we would use a similar process to how we forward propagated shallow neural networks, but now we will extend to more layers, calculating

$$\begin{cases} z^{[1]} = w^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} = g^{[1]}(z^{[1]}) \\ z^{[2]} = w^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} = g^{[2]}(z^{[2]}) \\ z^{[3]} = w^{[3]}a^{[2]} + b^{[3]} \\ a^{[3]} = g^{[3]}(z^{[3]}) \\ z^{[4]} = w^{[4]}a^{[3]} + b^{[4]} \\ a^{[4]} = g^{[4]}(z^{[4]}) \end{cases}. \tag{2.3.1}$$

In general, it is easy to see that

$$\begin{cases} z^{[\ell]} = w^{[\ell]}a^{[\ell-1]} + b^{[\ell]} \\ a^{[\ell]} = g^{[\ell]}(z^{[\ell]} \end{cases}, \tag{2.3.2}$$

for $0 \leq \ell \leq L$. For an entire training set, instead of a single training instance, we could use

$$\begin{cases} Z^{[\ell]} = w^{[\ell]}A^{[\ell-1]} + b^{[\ell]} \\ A^{[\ell]} = g^{[\ell]}(z^{[\ell]}) \end{cases}. \tag{2.3.3}$$

While we would prefer to never use for-loops, using a for loop to iterate over each layer is the best way to forward propagate.

2.3.2 Getting the matrix dimensions right

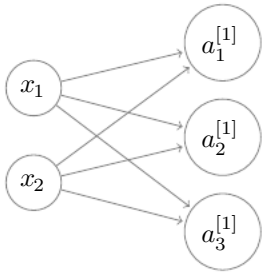


Figure 2.3.2: Figure for matix dimensions walkthrough

One of the aspects in deep learning that is easy to make a mistake on is calculating the dimensions of the matrices incorrectly. In this section, we are going to walk through how we can possibly think about the dimensions and make sure they are correct at each step. Consider the neural network in Figure 2.3.2, where our goal is to form $W^{[1]}$, $b^{[1]}$, and x in order to form

$$z^{[1]} = w^{[1]}x + b^{[1]}. \tag{2.3.4}$$

We know that input features correspond to the rows of our matrix and different training examples correspond to different columns. So, if each instance has two input features, that means each instance is in $\mathbb{R}^{2\times 1}$. Further, since $n^{[1]} = 3$, we know that for each instance, $z^{[1]} \in \mathbb{R}^{3\times 1}$. So, we currently have the dimensions for the equation to be

$$(3 \times 1) = w^{[1]}(2 \times 1) + b^{[1]}. \tag{2.3.5}$$

From matrix multiplication (without broadcasting), $w^{[1]}$ must have 2 columns to match the number of rows of x. We also know that when adding matrices the dimensions must match,

so $b^{[1]}$ must be in the same dimensional space as $z^{[1]}$ and $w^{[1]}x$, which means $b^{[1]} \in \mathbb{R}^{3\times 1}$. Now, from properties of matrix multiplication, $w^{[1]}x$ must have $w^{[1]} \in \mathbb{R}^{3\times 2}$. The matrix a will always be in the same dimensional space as z, since a only applies element operations to the elements of z. More generally, we can see that the sizes for each item in our neural network based on ℓ will be

$$z^{[\ell]}, a^{[\ell]} \in \mathbb{R}^{n^{[l]}\times m} \tag{2.3.6}$$

$$w^{[\ell]} \in \mathbb{R}^{w^{[\ell]}\times w^{[\ell-1]}} \tag{2.3.7}$$

$$b^{[\ell]} \in \mathbb{R}^{n^{[\ell]}\times 1}. \tag{2.3.8}$$

The derivatives of each variable will also be in the same dimensionsal space as the variables, so when we are backward propagating

$$dz^{[\ell]}, da^{[\ell]} \in \mathbb{R}^{n^{[l]}\times m} \tag{2.3.9}$$

$$dw^{[\ell]} \in \mathbb{R}^{w^{[\ell]}\times w^{[\ell-1]}} \tag{2.3.10}$$

$$db^{[\ell]} \in \mathbb{R}^{n^{[\ell]}\times 1}. \tag{2.3.11}$$

2.3.3 Why deep representations?

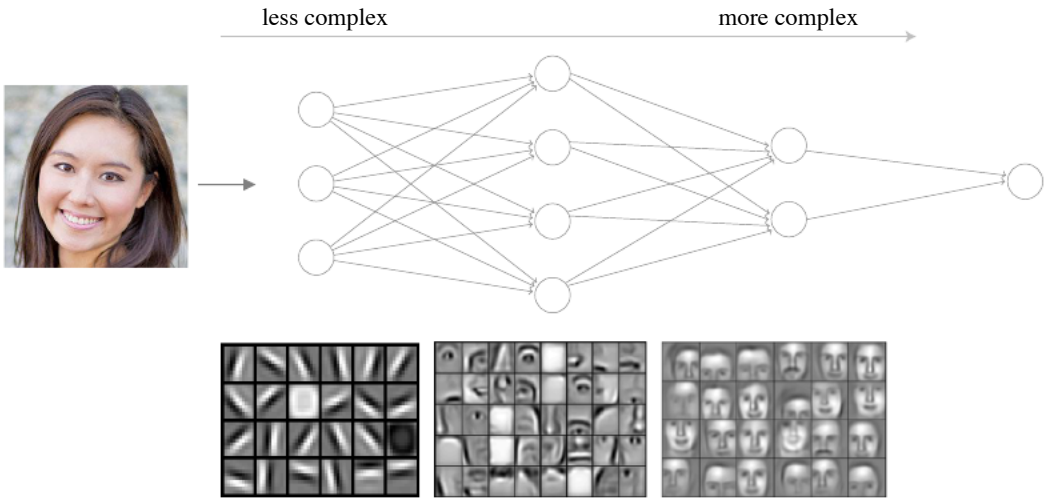


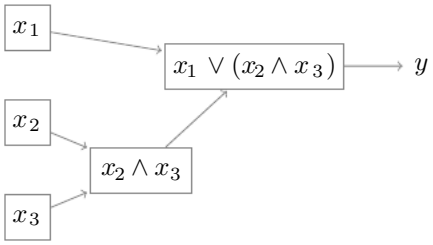
Figure 2.3.3: Deeper layers in a neural nexwork analyze more complex data

When using a neural network architecture, the deeper nodes look at more complex data. For example, with images, the first layer may detect edges, the second layer may put a few edges together, and the third layer may formulate the understanding of a face. Compounding knowledge throughout multiple layers is analogous to how many people believe the human brain works.

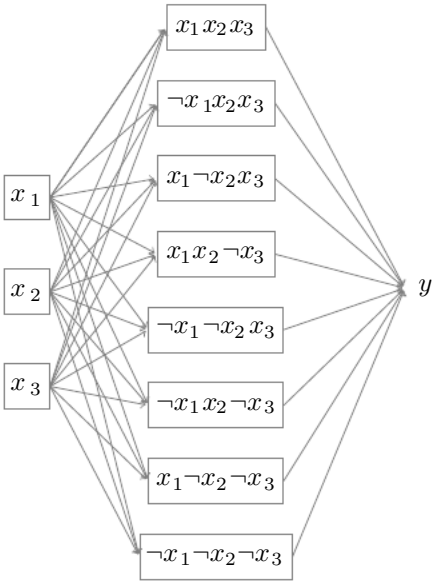
Graphs, in general, are often more useful when multiple layers can be stacked on top of each other. Consider the logic tree in Figure 2.3.4 that represents the function $y = x_1 \vee (x_2 \wedge x_3)$. With more layers, it is easier to store more information that becomes more complex as the layers grow in depth.

2.3.4 Forward and backward propagation

The forward propagation step on layer ℓ takes in as input $a^{[\ell-1]}$ and outputs $a^{[\ell]}$, which also caching $z^{[\ell]}$ for back propagation. For backward propagation, we will input $da^{[\ell]}$ and output $da^{[\ell-1]}, dW^{[\ell]}, db^{[\ell]}$. From the chain rule, we can derive the outputs of back propagation to



(a) Few nodes are required for a deeper trees



(b) Exponentially more nodes are required for a shallow tree, since it must test every possible combination of x_1, x_2, and x_3.

Figure 2.3.4: Comparing logic trees for $y = x_1 \vee (x_2 \wedge x_3)$ of different depth, it is easier to for deeper trees to represent the same information as shallow trees with less nodes.

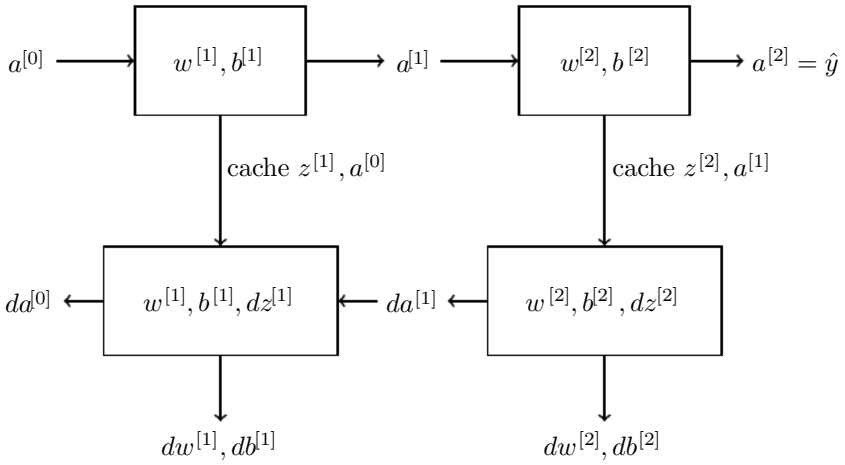


Figure 2.3.5: Forward and backward propagation steps for a two layer neural network. We first forward propagate across the top, while caching intermediate $z^{[\ell]}$ values, then we backward propagate across the bottom to find the changes to the weights.

give us the equations

$$\begin{cases} dz^{[\ell]} = da^{[\ell]} * g^{[\ell]\prime}(z^{[\ell]}) \\ dw^{[\ell]} = dz^{[\ell]} \cdot a^{[\ell-1]} \\ db^{[\ell]} = dz^{[\ell]} \\ da^{[\ell-a]} = w^{[\ell]T} \cdot dz^{[\ell]} \end{cases} . \tag{2.3.12}$$

For an entire training set, we would have

$$\begin{cases} dZ^{[\ell]} = dA^{[\ell]} * g^{[\ell]\prime}(Z^{[\ell]}) \\ dW^{[\ell]} = {}^{1}/_{m} \cdot dZ^{[\ell]} \cdot A^{[\ell-1]T} \\ db^{[\ell]} = {}^{1}/_{m} \cdot \text{np.sum}(dZ^{[\ell]},\ \text{axis=1, keepdims=True}) \\ dA^{[\ell-1]} = W^{[\ell]T} \cdot dZ^{[\ell]} \end{cases} . \tag{2.3.13}$$

2.3.5 Parameters vs hyperparameters

For neural network models, our **parameters** consist of strictly weights $W^{[\ell]}$ and bias $b^{[\ell]}$ parameters. However, there are plenty more variables that we have to set in order to optimize our function, such as the learning rate α, number of gradient descent iterations, number of hidden layers L, number of hidden units $n^{[\ell]}$, and type of activation function. The variables that affect our parameters, as just listed, are known as **hyperparameters**. Many hyperparameters are set on an experimental basis and there are many uses of neural networks that determine optimal hyperparameters.

2.3.6 Connection to the brain

Modern deep learning has little resemblance to what we know about how the brain learns. While it is still unknown how a neuron in the brain works, we have a loose idea that each neuron takes some signal and passes it to the next signal. Putting many neurons together then can sort of resembles a tree-like structure similar to a neural network. However, there is little evidence that anything like forward and backward propagation occurs, so the connection to the brain is very loose.

Chapter 3

Optimizing and structuring a neural network

3.1 Full-cycle deep learning projects

For most machine learning projects we have to do the following tasks in order:

1. find a problem
2. get data
3. design a model
4. train the model
5. test the model
6. deploy the project
7. maintain the project

The original training model that we choose tends to be changed in order to find the best model that performs best. Most machine learning papers tend to focus on steps 2-5, but in this section, we will focus on steps 1, 6, and 7. For deep learning problems, good problems for us would tend to be interesting with available data in a field we have domain knowledge in, as well as useful and feasible to create. Feasibility is a massive problem for choosing projects, especially in industry. To find data let us use the example of a trigger word speech detection system, such as *alexa*. It may be a good idea to first use a small set of data in order to quickly start building out a model and finding out the difficulties in the problem at hand; even a couple hundred instances may initially work well. One way to collect new data would be to go around and ask different people to speak into a recording device, where they say either nothing, *alexa*, or some other words.

As we train the models iteratively, it is a good idea to document each model that we set up and the results it performed. That way, it will be easy to look back on older models when making tweaks to a current system.

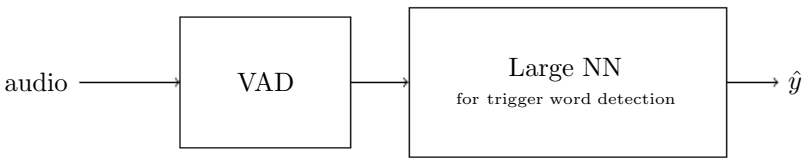


Figure 3.1.1: Breaking a problem involving trigger word detection into two parts: voice activity dection (VAD) and trigger word detection.

For our system involving a trigger word detection, it may be a good idea to break the problem into two parts: one that detects if a word is spoken, and one that detects if the

spoken word was the trigger word. In modern literature, the word speaking detector may be referred to as voice activity detection (VAD). When using VAD we primarily have two options to choose from when selecting a model. We can use a non-machine learning based approach which detects if the volume is greater than some value ϵ, or we can use a small neural network that is trained to detect human speech. Since the non-machine learning option may only take a few minutes to implement, it may be best to choose that option first in order to minimize cost and move quickly. Another downside to the neural network approach is that new accents may be introduced in production that the model has never heard before which could cause abnormal results.

3.2 Setting up a machine learning application

3.2.1 Training, development, and testing sets

Applied machine learning involving choosing many hyperparameters for a model, such as the number of layers, hidden units, learning rates, and activation functions. Most of the choices of hyperparameters for one subset of problems, such as natural language processing, tend to not work very well on other machine learning problems, like computer vision. Because it is often difficult to determine which hyperparameters work best, we tend to check our choice of hyperparameters against some test training set in order to determine its effectiveness.

Figure 3.2.1: A data set is broken into three parts: training, development, and testing.

We tend to break our data into three sets: training, cross-validation or development, and testing.[1] The training data set is used to change the weights and bias of our model, the cross-validation set is used to determine if our model has overfitted the data and determine if our hyperparameters generalize well to data we have not seen, and the testing set is used to calculate how accurate our model will be once deployed. In the past, the best practice for breaking the data up was to give the training set $\sim 60\%$ of the data, the development set $\sim 20\%$ of the data and the testing set $\sim 20\%$ of the data. Now, however, our datasets have grown large enough to the point that breaking up the data based on percentages no longer makes sense. Today, we may break up a data set with 1 million examples into a development set with 10 thousand examples and a testing set with 10 thousand examples. It is important to distribute different types of data between the testing and development set evenly.

3.2.2 Bias and variance

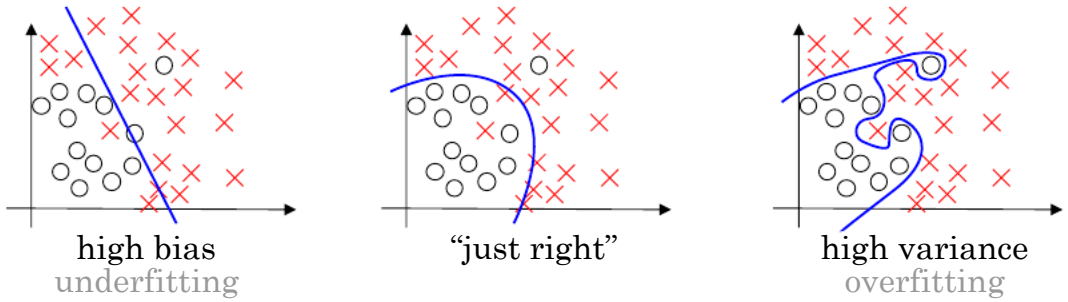


Figure 3.2.2: Different types of variance and bias.

When we talk about the **variance** for a particular model, we are referring the amount of overfitting in our model. When there is a lot of overfitting, we say the model has high

[1]Many people may only use a training and testing set, where the testing set acts as if it were a development set.

variance. similarly, when the model is underfitting, we say there is low variance. We use the term **bias** to describe how well our model fits the data. If our model performs well on both the training and development set, we say that the model has low bias, and if the model performs badly on both the training and development set, we say the model has a high bias. We often determine variance based on the training set and development set error rates.

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	high variance	high bias	high bias high variance	low bias low variance

Table 3.1: How we may denote high and low variance and bias when given the training and development set error rates. Here, we assume that the human error is approximately 0%.

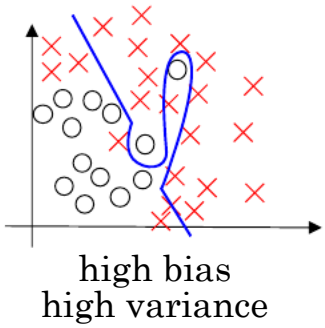


Figure 3.2.3: An example of a model having both high variance and high bias.

3.2.3 Combatting bias and variance

With a model, we first check for high bias and then high variance. When a model has **high bias** the network will not be generalizing well on the training data. It may be a good idea to test out a larger neural network, run gradient descent longer, or change neural network architectures. When a model has **high variance**, the development set error will be high. We can improve the high variance problem by introducing new data, regularization, or changing the neural network architecture.

Before neural networks became widely adopted, we often had to make decisions on whether or not to improve the bias *xor* variance. Having to choose one or the other came to be known as the **bias variance tradeoff**. However, with larger neural networks we have been able to independently improve both the bias and variance of a network.

3.3 Regularizing a neural network

3.3.1 Regularization

Regularization will be a way to improve a high variance model. To show why regularization may be useful, let us consider the logistic regression where our goal was to find

$$\min_{w,b} J(w,b) \tag{3.3.1}$$

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

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right). \tag{3.3.2}$$

When using regularization we will redefine our cost function to be

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}||w||_2^2, \tag{3.3.3}$$

where we are using the L2 norm defined as

$$||w||_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w. \tag{3.3.4}$$

When using the L2 norm in our regularization term, we say that we are using **L2 regularization**. The other main type of regularization is **L1 regularization**, where our regularization term is

$$\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}||w||_1. \tag{3.3.5}$$

L1 regularization tends to result in w being a sparse vector, which is a vector with a lot of 0s and convenient for calculations. L2 regularization tends to be used a lot more often with neural networks. The term λ is known as the **regularization parameter** which is another hyperparameter often set during the validation process of training the network.

For neural networks, we want to find

$$\min_{w^{[1]},b^{[1]},\ldots,w^{[L]},b^{[L]}} J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}), \tag{3.3.6}$$

where our cost function is also defined as

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right). \tag{3.3.7}$$

The regularization involving multiple bias and weight terms is

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{\ell=1}^{L}||w^{[\ell]}||_F^2, \tag{3.3.8}$$

where the squared norm of a matrix is known as the **Frobenius norm**, which is defined as

$$||w^{[\ell]}||_F^2 = \sum_{i=1}^{n^{[\ell-1]}}\sum_{j=1}^{n^{[\ell]}}\left(w_{ij}^{[\ell]}\right)^2. \tag{3.3.9}$$

Before we added the additional regularization term to our cost function, we updated the weights like

$$w^{[\ell]} = w^{[\ell]} - \alpha\, dw^{[\ell]}, \tag{3.3.10}$$

where we calculated $dw^{[\ell]}$ to be the change in the cost function with respect to w. When adding the regularization term, our calculation of gradient descent must also include changes to the regularization term with respect to the weights and bias. When our bias changes, the regularization term does not change, so the gradient descent term for the calculation of bias remains the same. However, when our weights change, the derivative of our regularization term turns out to be

$$\frac{d}{dw^{[\ell]}}\left[\frac{\lambda}{2m}||w^{[\ell]}||_F^2\right] = \frac{\lambda}{m}w^{[\ell]}. \tag{3.3.11}$$

While it would seem weird that the derivative of the Frobenius norm results in a matrix (since the Frobenius norm produces a scalar value), we are really just looking for the changing result of the Frobenius norm due to changes in the entire matrix and not the changes due to one element, which is why the result ends up as a matrix. Now we can rewrite our weights gradient descent expression as

$$w^{[\ell]} = w^{[\ell]} - \alpha\left(dw^{[\ell]} + \frac{\lambda}{m}w^{[\ell]}\right). \tag{3.3.12}$$

Distributing the alpha term, we have

$$w^{[\ell]} = w^{[\ell]} - \alpha\, dw^{[\ell]} - \alpha \frac{\lambda}{m} w^{[\ell]}. \tag{3.3.13}$$

Notice that with the terms

$$w^{[\ell]} - \alpha \frac{\lambda}{m} w^{[\ell]} \tag{3.3.14}$$

in our gradient descent function, the addition of the regularization term will always decrease the values of the weights, since α, λ, and m are all nonnegative. We sometimes refer to regularization that strictly decreases the weights as **weight decay**.

3.3.2 Why regularization reduces overfitting

With the L2 regularization of the cost function

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m} \sum_{\ell=1}^{L} ||w^{[\ell]}||_F^2, \tag{3.3.15}$$

making λ a large number will make the term

$$\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) \tag{3.3.16}$$

negligible in comparison. In the scenario where λ is very large, we must make the weights very small in order to minimize the cost. If the weights are near zero, they will not have any effect on the neural network and our new neural network will become much simpler. In practice, however, all of the weights become much smaller, not just a select few as shown in Figure 3.3.1. But, with all the weights near 0, the model will still become simpler.

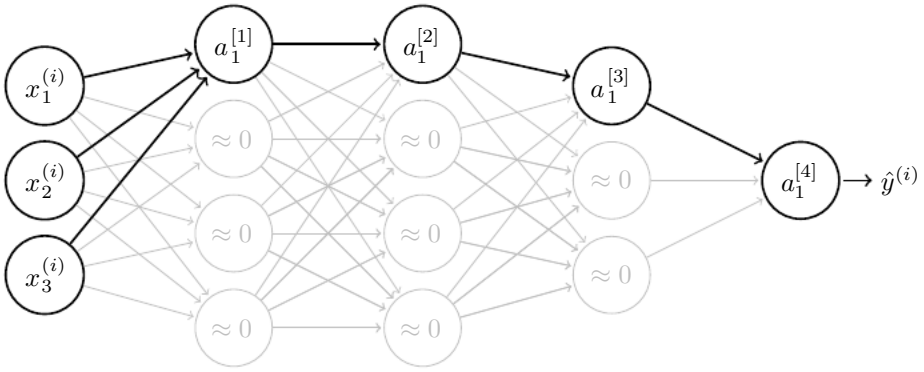


Figure 3.3.1: If the weights connected to a neuron are near 0, then the neuron will not make a significant contribution to the hypothesis. With less neurons in the neural network, the model will become simpler.

With our linear activation calculation of

$$z^{[\ell]} = w^{[\ell]} a^{[\ell-1]} + b^{[\ell]}, \tag{3.3.17}$$

if we make $w^{[\ell]}$ really small, then $z^{[\ell]}$ will end up really small (assuming $b^{[\ell]}$ does not make a significant contribution). If we are using the hyperbolic tangent function or the sigmoid function as our activation function, then when z is small we will pretty much have a linear activation function. Recall that with a linear activation function, the depth of our neural network would not matter and the entire model could just be represented with a linear function.

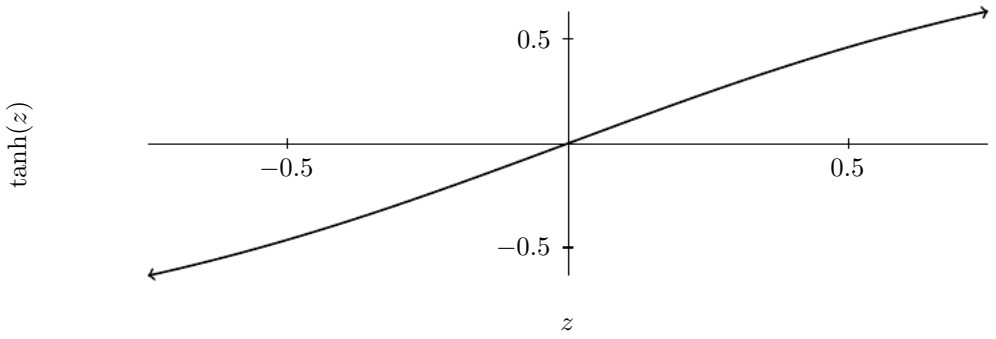


Figure 3.3.2: The hyperbolic tangent function is nearly a linear function for values near 0

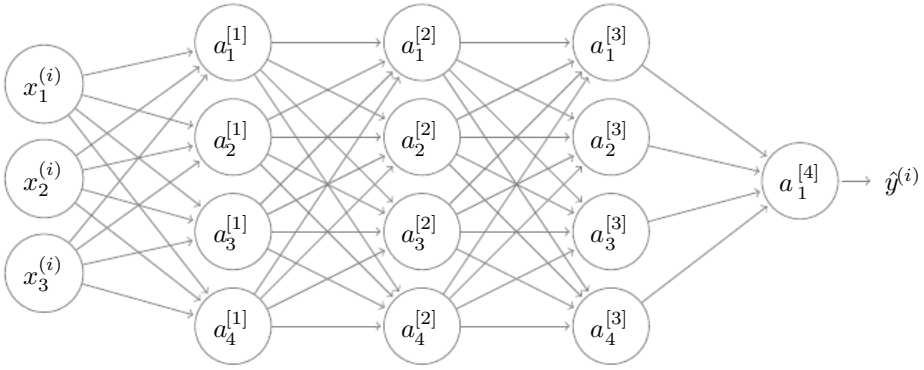


Figure 3.3.3: 4 layer neural network with 3 hidden layers each with 4 neurons

3.3.3 Dropout regularization

Consider the neural network in Figure 3.3.3, which has a total of 12 hidden neurons. For each iteration of gradient descent that we train our neural network on, dropout regularization visits each hidden neuron and randomly chooses whether or not that neuron will be removed from the neural network. The probability that each hidden neuron remains is a hyperparameter that we can set. With a neural network has a differing number of neurons in each hidden layer, we can make the *keep_prob* value a hyperparameter in each layer. If a neuron is removed, then all the weights and bias connected to the neuron are also removed. When testing our neural network or once it is deployed, we will no longer use dropout regularization, primarily because we do not want randomized predictions. In computer vision, the input to the neural network is very large and many people in computer vision use dropout regularly.

To implement dropout regularization let us set

$$d\ell = \text{np.random.rand}(a\ell.\text{shape}[0],\, a\ell.\text{shape}[1]) < keep\_prob, \tag{3.3.18}$$

where *keep_prob* is the probability that we keep the neuron and $d\ell$ is the same shape as $a^{[\ell]}$. $d\ell$ will now store a matrix of True or False values. We can then take

$$a\ell = a\ell * d\ell \tag{3.3.19}$$

in order to zero out the values in the matrix where $d\ell$ is 0 and keep the values in the matrix where $d\ell$ is 1. We also want to make sure that $z^{[\ell]}$ is not reduced from its expected value without dropbox regularization, which can be done by taking

$$a\ell = a\ell / keep\_prob. \tag{3.3.20}$$

Consider, for example, if we had 100 hidden neurons in a layer and we set $keep\_prob = 0.8$. On average, 20 neurons would be hidden and $a^{[\ell]}$ would be reduced by 20%. In order to

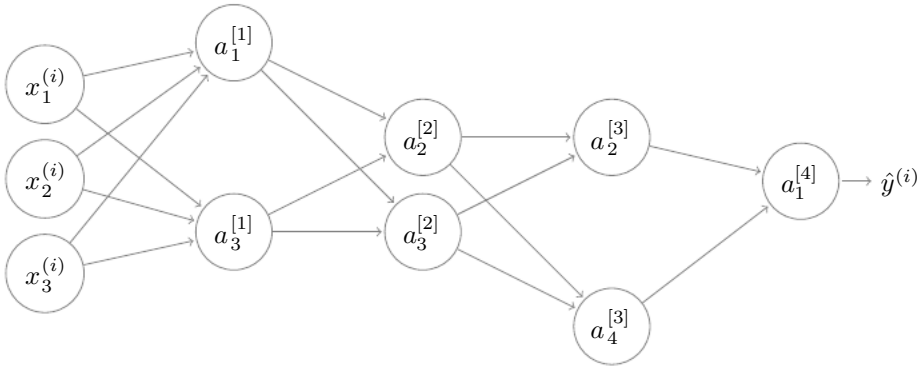


Figure 3.3.4: A potential simplified neural network with only the hidden neurons that are not removed in dropout regularization.

mitigate this loss, divide by our *keep_prob* value of 0.8. Dividing by the hyperparameter *keep_prob* is known as the **inverted dropout** technique.

With dropout regularization, any feature could be removed randomly from a neural network on any given iteration. Because of this, the network can learn not to rely on any given feature too much, which can regularize a model. In general, when using dropout regularization, the weights shrink in size.

3.3.4 Other regularization methods

While we know that adding more data to our model could help reduce overfitting, we do not always have easy access to new data. Instead, one way to add more data is to augment the existing data that we have, which is known as **data augmentation**. If we were to augment images, for example, then we might reflect, crop, distort, and change the colors of our inputted data as shown in Figure 3.3.5.

Figure 3.3.5: Data augmentation on images

Early stopping is another way to regularize our model. With early stopping, we decrease the number of iterations when running gradient descent in hopes of finding a more general model. When using early stopping, the primary downside is that we are no longer looking

to completely optimize the cost function and it is often hard to determine when it is a good time to stop iterating.

3.4 Setting up an optimization problem

3.4.1 Normalizing inputs

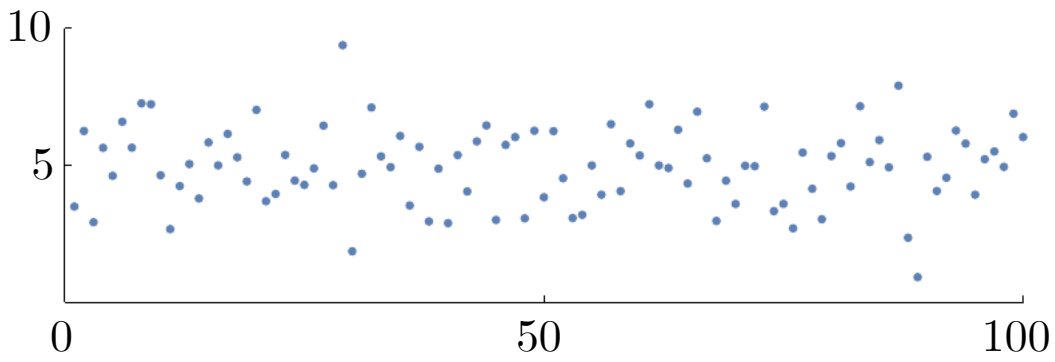


Figure 3.4.1: Normal distribution of data vertically centered at 5 with a variance of 2.25.

To normalize a dataset, we must do two things: center the mean at the origin in each direction and set the variance to be 1. Consider the two dimensional data in Figure 3.4.1. Centering the data in each direction would require us to calculate each direction's mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \tag{3.4.1}$$

and then take each point and subtract the mean, which can be done with

$$x = x - \mu. \tag{3.4.2}$$

The result after centering the data in each direction at 0 can be shown in Figure 3.4.2.

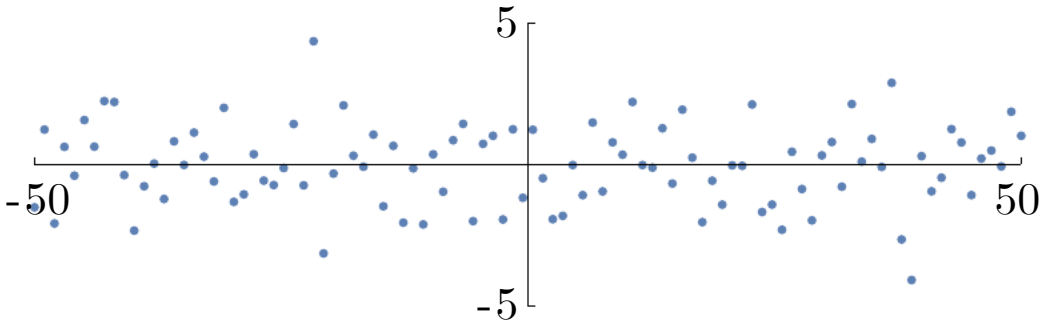


Figure 3.4.2: Centering the mean of the data in each direction at 0

The variance is defined as the square of the standard deviation. In general, the variance is

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)^2, \tag{3.4.3}$$

but since we have set $\mu = 0$, we can simpifly the variance to

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)}\right)^2. \tag{3.4.4}$$

To normalize the variance, we must set the variance of the entire dataset to be 1, which can be done by dividing the entire dataset by the standard deviation

$$x = \frac{x}{\sigma}. \tag{3.4.5}$$

Figure 3.4.3 shows the result after making the variance of the data 1 and displays the normalized data. It is important when using a training set and testing set that both sets

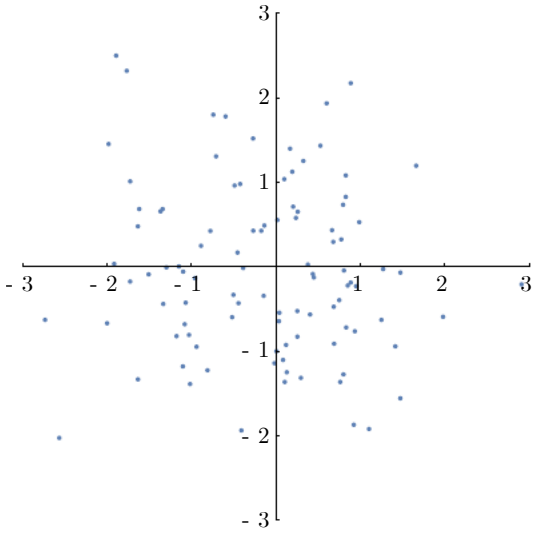
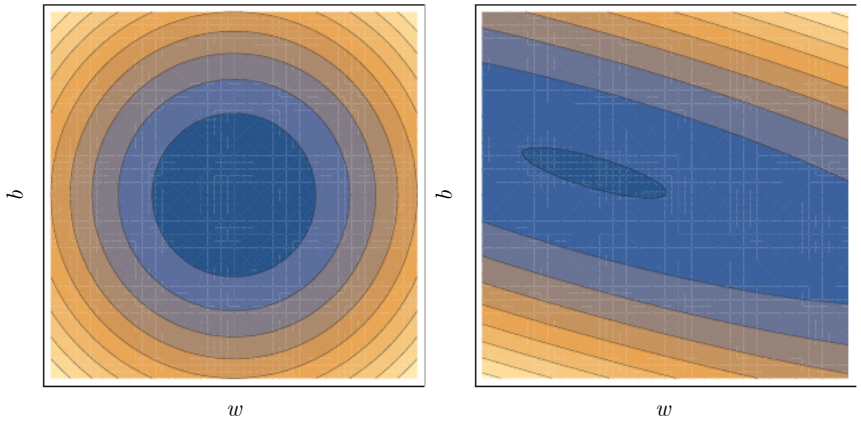


Figure 3.4.3: Normalized data after centering the mean at 0 and making the variance 1.

are normalized with the same μ and σ^2 in order to best simulate how the model would perform when deployed.



(a) Normalized **(b)** Unnormalized

Figure 3.4.4: Potential contour plots of the cost function for normalized and unnormalized data.

Normalizing the data often makes it quicker for gradient descent to converge. Considering the two-dimensional contour plots of a potential cost function in Figure 3.4.4, our goal with normalizing is to make it easier to find the minimum of the cost function. If we have a set of features with a range of $(-1000, 1000)$ and another set of features with a range of $(-1, 1)$, it is often quite hard to pick a learning rate that will work well for both features.

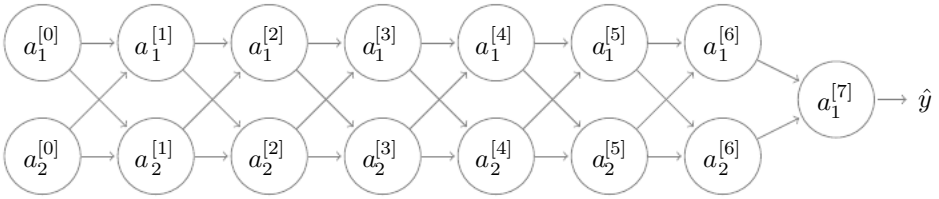


Figure 3.4.5: 7 layer neural network

3.4.2 Data vanishing and exploding gradients

Consider the neural network in Figure 3.4.5, where we will assume the input is non-negative and will ignore the bias term for now. Here, we will exclusively use ReLU activations and initialize all of our weights to be in the form of

$$W^{[\ell]} = \begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix}, \tag{3.4.6}$$

where $\gamma > 0$. In the case presented, the ReLU activation function will behave like a linear activation function and we will have

$$W^{[7]}W^{[6]}W^{[5]}W^{[4]}W^{[3]}W^{[2]}\,W^{[1]}\,a^{[0]} = \hat{y}. \tag{3.4.7}$$

The multiplication of all of the $W^{[\ell]}$ will simplify to

$$\prod_{\ell=1}^{7} W^{[\ell]} = \prod_{\ell=1}^{7} \begin{bmatrix} \gamma & 0 \\ 0 & \gamma \end{bmatrix} = \begin{bmatrix} \gamma^7 & 0 \\ 0 & \gamma^7 \end{bmatrix}. \tag{3.4.8}$$

When we initialize $\lambda < 1$ we will end up multiplying our input by an exponentially small number and when we initialize $\lambda > 1$ multiplication will be exponentially large. Consider, for example, if we set $\lambda = 0.5$ then our input will be scaled down by a factor of $\lambda^7 = 0.0078125$. similarly, if we set $\lambda = 1.5$ then the input will be scaled up by $\lambda^7 = 17.0859375$. When the prediction of our model is a huge number, then the gradients in back propagation will *explode* and when our prediction is near zero, the data will have a negligible impact.

3.4.3 Weight initialization for deep neural networks

For each neuron, we calculate the linear activation to be

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n. \tag{3.4.9}$$

At the moment, we are going to ignore the bias term. Notice that when $n \gg 0$, we would like the weights to be small in order to keep z from exploding. similarly, when n is near 0, making the weights larger would make the most sense in order to have the input make a contribution. One solution is to initialize our weights to have a variance of

$$\sigma^2 = \frac{1}{n}. \tag{3.4.10}$$

We previously calculated the variance of a data set centered at zero to be

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)}\right)^2. \tag{3.4.11}$$

Starting with a random normally distributed data set with a variance of $\sigma^2 = 1$, we can change the variance to be ζ by multiplying each instance by $\sqrt{\zeta}$. We can show that this is correct by carrying out the multiplication on each instance and using substituting with the original variance of 1, which gives us

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)}\sqrt{\zeta}\right)^2 \tag{3.4.12}$$

$$= \zeta\left(\frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)}\right)^2\right) \tag{3.4.13}$$

$$= \zeta. \tag{3.4.14}$$

So, if we wanted to make the variance $1/n$, we would initialize our weights to be

$$W^{[\ell]} = \text{np.random.randn}(shape) * \text{np.sqrt}\left(\frac{1}{n^{[\ell-1]}}\right). \tag{3.4.15}$$

The modern best practice tends to set the variance of weights with the ReLU function to be

$$\sigma^2 = \frac{2}{n^{[\ell-1]}}, \tag{3.4.16}$$

which is known as **He initialization**. With a hyperbolic tangent function we generally stick with

$$\sigma^2 = \frac{1}{n^{[\ell-1]}}, \tag{3.4.17}$$

which is known as **Xavier initialization**. Occasionally, some people may also set the variance to be

$$\sigma^2 = \frac{2}{n^{[\ell-1]} + n^{[\ell]}}. \tag{3.4.18}$$

3.4.4 Gradient numerical approximations

Numerical approximations of our gradient can be useful when debugging a network and testing if the forward and backward propagation steps are working correctly. From the definition of the derivative, we know that the one-sided limit of

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta)}{\epsilon} \tag{3.4.19}$$

will give us the slope of the line tangent to a two variate equation centered at θ. We could also use use the two sided limit

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \tag{3.4.20}$$

to calculate the derivative, which will produce an equivalent result. The two-sided limit as shown in Figure 3.4.6 works best for numerical approximations becuase it gets a more accurate picture of the slope at both sides.

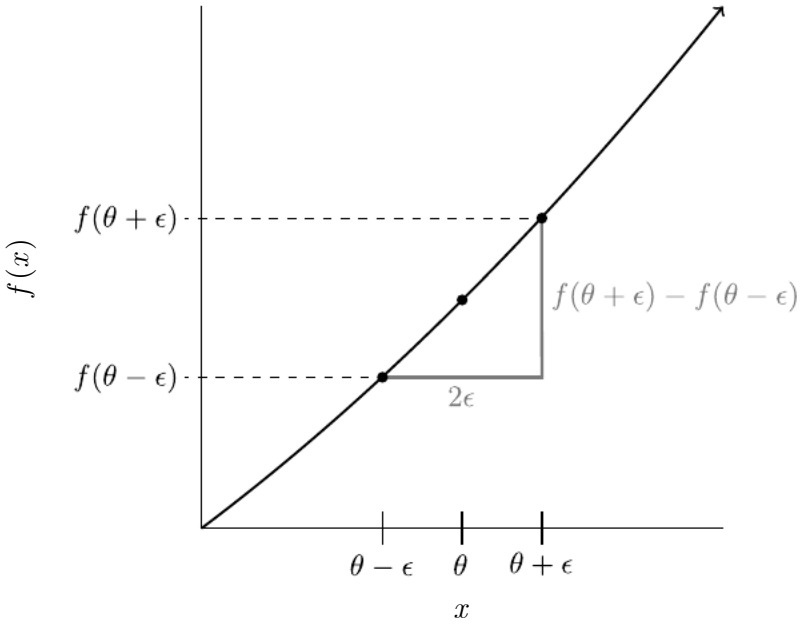


Figure 3.4.6: Two-sided limit derivative definition

The error when using the two-sided limit is $O(\epsilon^2)$ and the error when using a one-sided limit is $O(\epsilon)$. Although it is twice as computationally expensive, it is often worth it to use the two-sided limit.

3.4.5 Gradient checking

Using the numerical approximations of the derivative our goal is to check if our calculations implemented in back propagation are correct. To simplify calculations, we will store each entry in all of our parameters $W^{[1]}$, $b^{[1]}$, ..., $W^{[L]}$, $b^{[L]}$ into a single large vector denoted as θ. similarly, we will store $dW^{[1]}$, $db^{[1]}$, ..., $dW^{[L]}$, $db^{[L]}$ into a vector $d\theta$. We now can write our cost function as

$$J(\theta) = J(\theta_1, \theta_2, \theta_3, \dots). \tag{3.4.21}$$

For each element i of our vector θ, we can approximate the derivative with

$$d\tilde{\theta}_i = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}. \tag{3.4.22}$$

If our calculations are correct, then $\tilde{d\theta}$ should approximately be $d\theta$. We can check our error rate E by taking the normalized Euclidean distance between the vectors of

$$E = \frac{\left\| \tilde{d\theta} - d\theta \right\|_2}{\left\| \tilde{d\theta} \right\|_2 + \left\| d\theta \right\|_2}. \tag{3.4.23}$$

In practice, if we set $\epsilon = 10^{-7}$, then with $E < 10^{-7}$ we can generally confirm that our calculations are correct and with $E > 10^{-3}$ we should double check our calculations. For values in-between, it is harder to determine correctness. For debugging, it is best to first check for bugs where the difference between $d\tilde{\theta}_i$ and $d\theta_i$ are the greatest.

We can run gradient checking on L2 regularization by making sure to include our regularization term in the cost function. However, because of the randomness of dropout regularization, we cannot use gradient checking with dropout on.

3.5 Optimization algorithms

3.5.1 Mini-batch gradient descent

Deep learning works best when there is a lot of data and we can quickly optimize our model. We have previously been using gradient descent on all of our training examples with our vectorized input

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix}, \tag{3.5.1}$$

where $X \in \mathbb{R}^{n_x \times m}$ and our vectorized output

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}, \tag{3.5.2}$$

where $Y \in \mathbb{R}^{1 \times m}$. The gradient descent we have previously been using is known as **batch gradient descent**. If $m \gg 0$, then it may take a while to compute each iteration of gradient descent. For now, suppose that our training set is of size $m = 5,000,000$. Instead of using all m training examples for each iteration of gradient descent, let us break the input into groups of 1000, which we will call **mini-batches**. For example, the first mini-batch will be have

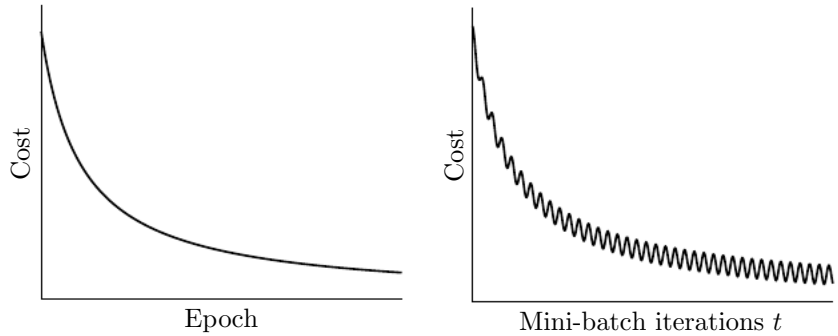
$$\begin{cases} X^{\{1\}} = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(1000)} \end{bmatrix} \\ Y^{\{1\}} = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(1000)} \end{bmatrix} \end{cases} \tag{3.5.3}$$

where we will denote the ith mini-batch with superscript $\{i\}$. In total, we will have 5000 mini-batches. Next, we will make a pass through the training set as described in Algorithm 3. Each pass through the training set using any form of gradient descent is known as an **epoch**.

With batch gradient descent, our cost will ideally always be decreasing, however, that is not the case with mini-batch gradient descent because each mini-batch does not reflect the entire training set and our gradient may not step in the perfect direction that minimizes our cost.

Algorithm 3 A single pass through the training set using **mini-batch gradient descent**.

for $t \in \{1, \ldots, 5000\}$ **do**
  Forward propagation on $X^{\{t\}}$
  Compute the cost for $J^{\{t\}}$
  Backprop to compute gradients
  Update the weights and bias
end for



(a) Batch gradient descent **(b)** Mini-batch gradient descent

Figure 3.5.1: How the cost function could decrease based on the choice of gradient descent.

When setting the hyperparameter that represents the size of the mini-batch, there are two extreme cases. First, if we set the mini-batch size equal to m, then we just have our old method of batch gradient descent. If we set the mini-batch size to be 1 then each example is a mini-batch; we call this **Stochastic gradient descent**. With Stochastic gradient descent, we cannot use vectorization and will not typically converge to a single value, instead, it will jump around near the minimum. In practice, we often choose a mini-batch size in-between the two extremes in order to preserve vectorization and make faster progress.

The modern best practice in the field is to use batch gradient descent when $m \leq 2000$, otherwise, we typically use 2^6, 2^7, 2^8 or 2^9 in order to store data most efficiently.

3.5.2 Exponentially weighted averages

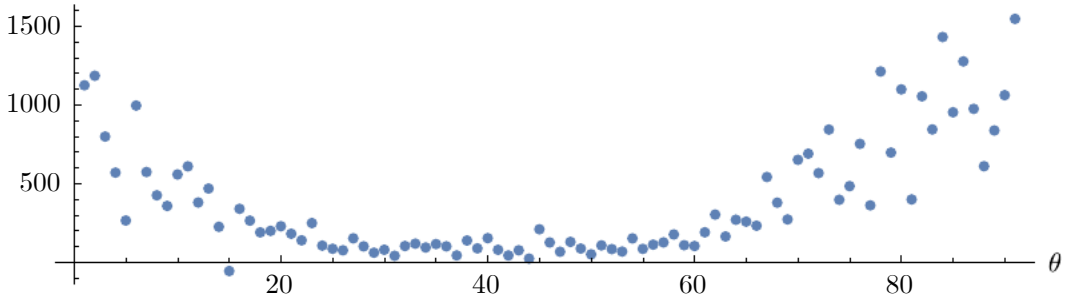


Figure 3.5.2: Data set to calculate the exponentially weighted average

To describe other optimization algorithms that may work better than gradient descent, we first need to introduce the concept of exponentially weighted averages. Consider the dataset

in Figure 3.5.2 where

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{100} \end{bmatrix} = \begin{bmatrix} 1124 \\ 1184 \\ \vdots \\ 1546 \end{bmatrix}. \tag{3.5.4}$$

To calculate the exponentially weighted average, we will start by initializing $v_0 = 0$ and recursively defining

$$v_t = 0.9\, v_{t-1} + 0.1\, \theta_t \tag{3.5.5}$$

for $t \in \{1, \ldots, 100\}$. After computing all of the v_t points, we can plot the joined data as shown in red in Figure 3.5.3.

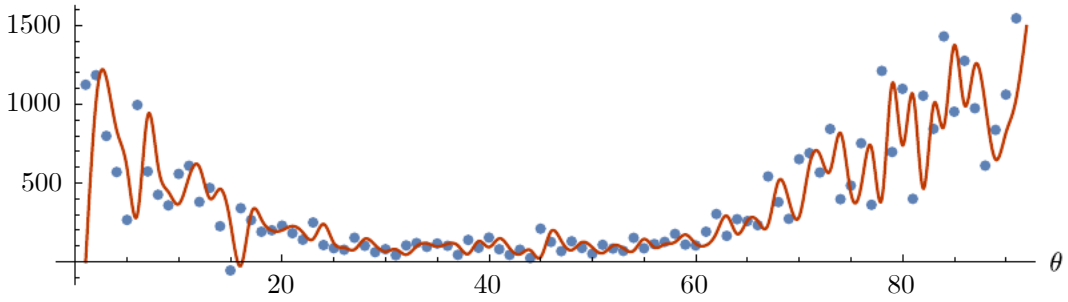


Figure 3.5.3: Exponentially weighted average as a joined plot (red)

In general, we can set

$$v_t = \beta\, v_{t-1} + (1 - \beta)\, \theta_t, \tag{3.5.6}$$

where $0 \leq \beta < 1$. The choice of β sets the value of v_t to approximately be the average over the last

$$\frac{1}{1-\beta} \tag{3.5.7}$$

data points. For example, when we set $\beta = 0.9$, we were closely mirroring each value of v_t to be the average of the previous 10 instances. Additional changes to the value of β are shown in Figure 3.5.4. In statistics, the exponentially weighted average may also be referred to as the **exponentially weighted moving average**.

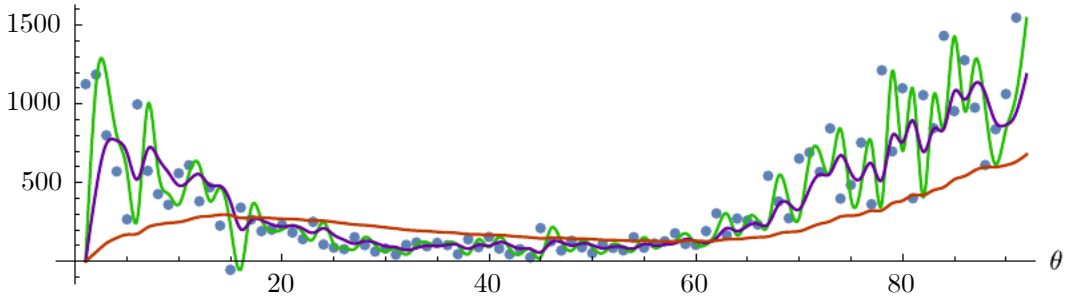


Figure 3.5.4: $\beta = 0.01$ (green), $\beta = 0.6$ (purple), $beta = 0.95$ (orange)

To better understand exponentially weighted averages, let us set $\beta = 0.9$ and look at the sequence

$$\begin{cases} v_{100} = 0.9\, v_{99} + 0.1\, \theta_{100} \\ v_{99} = 0.9\, v_{98} + 0.1\, \theta_{99} \\ v_{98} = 0.9\, v_{97} + 0.1\, \theta_{98} \\ \vdots \end{cases}. \tag{3.5.8}$$

Using substitution, we can write

$$v_{100} = 0.9\,(0.9\,(0.9\,(v_{97}) + 0.1\,\theta_{98}) + 0.1\,\theta_{99}) + 0.1\,\theta_{100} \tag{3.5.9}$$

$$= 0.1\,\theta_{100} + (0.1 \times 0.9)\theta_{99} + \left(0.1 \times 0.9^2\right)\theta_{98} + \left(0.1 \times 0.9^3\right)\theta_{97} + \cdots \tag{3.5.10}$$

$$= \sum_{t=1}^{100} \left(0.1 \times 0.9^{100-t}\right)\theta_t. \tag{3.5.11}$$

Visually, the the weight that is contributed to the θ_{100} term decreases exponentially as shown in Figure 3.5.5.

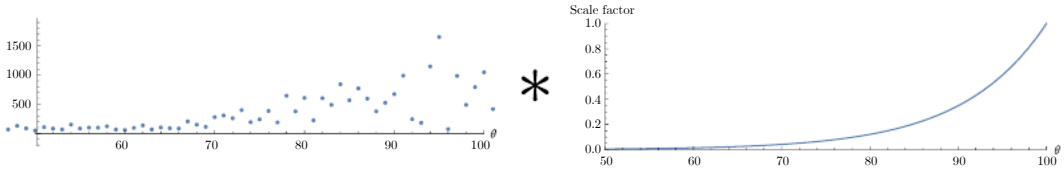


Figure 3.5.5: At θ_{100} is the sum of the element-wise products between the scaling factors and data

The time constant τ is the time it takes for an exponentially decaying plot to reach $1/e$ its max. From our decay equation, we have

$$\beta^{t-\tau} = \frac{1}{e}. \tag{3.5.12}$$

Solving for τ, we find the time constant to be

$$\tau = t + \log_\beta(e), \tag{3.5.13}$$

where $\log_\beta(e)$ represents the number of data points we go back in order to calculate $1 - 1/e$ of the value of v_t. The approximation in Equation 3.5.6 is a simplified version of $-\log_\beta(e)$ and is often used because it gives a close enough answer to the true time constant in our domain for β. The differences in $-\log_\beta(e)$ and $1/1-\beta$ are shown in Figure 3.5.6.

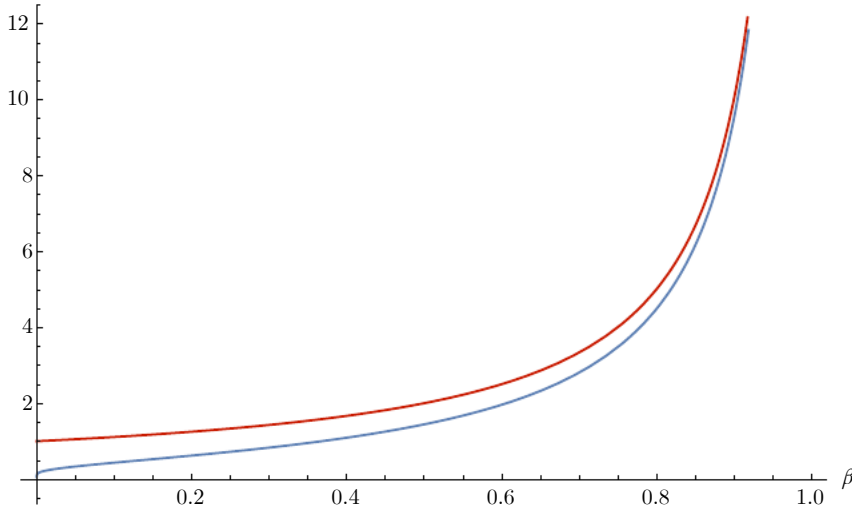


Figure 3.5.6: Approximation of the time constant $\tau = 1/1-\beta$ (red) and the real time constant $\tau = \log_\beta(e)$ (blue)

We generally choose to use this definition of a weighted average instead of truly calculating the average in a particular window because it is easier to compute and requires less memory storage.

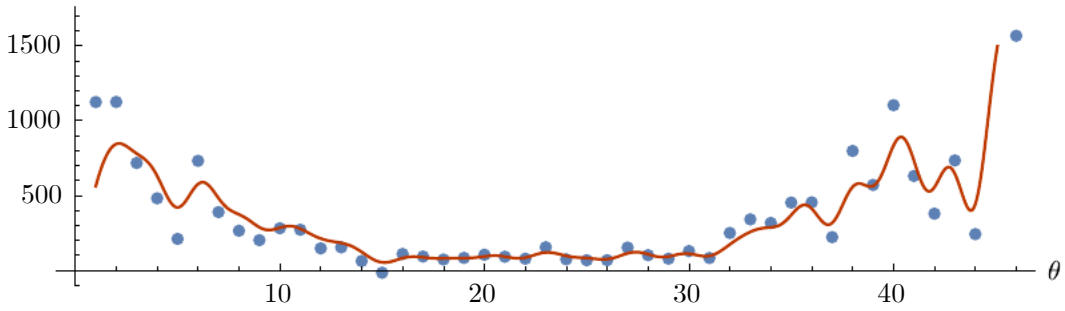


Figure 3.5.7: Data for bias correction analysis, where the red line is the exponentially weighted average when $\beta = 0.5$

3.5.3 Bias correction for exponentially weighted averages

Notice that at the beginning period of the plot in Figure 3.5.7, the plot starts off with pretty small values and grows relatively slow. Let us denote $\epsilon = 1 - \beta$. Because of how we defined our exponentially weighted average, each plot will start with

$$\begin{cases} v_0 = 0 \\ v_1 = \theta_1 \epsilon \\ v_2 = \beta\theta_1 \epsilon + \theta_2 \epsilon \\ v_3 = \beta^2 \theta_1 \epsilon + \beta\theta_2 \epsilon + \theta_3 \epsilon \end{cases} . \tag{3.5.14}$$

We can more easily interpret this data by plugging in a few values of β and analyzing the results, which is done in Table 3.2.

β	0.1	0.5	0.9
v_1	$0.9\,\theta_1$	$0.5\,\theta_1$	$0.1\,\theta_1$
v_2	$0.09\,\theta_1 + 0.9\,\theta_2$	$0.25\,\theta_1 + 0.5\,\theta_2$	$0.09\,\theta_1 + 0.1\,\theta_2$
v_3	$0.009\,\theta_1 + 0.09\,\theta_2 + 0.9\,\theta_3$	$0.125\,\theta_1 + 0.25\,\theta_2 + 0.5\,\theta_3$	$0.081\,\theta_1 + 0.09\,\theta_2 + 0.1\,\theta_3$

Table 3.2: Calculations of the first three v_t terms.

Notice that when β is large, v_t starts off growing the slowest. If we set each value of θ_n to 1 then we can get the total amount we are scaling our input. We would expect the scaling factor to be 1 in the general case because when averaging n numbers, if each number is 1, then the average is 1. However, when $\beta = 0.9$ our total scaling factor for v_1 is 0.1, for v_2 is 0.19, and for v_3 is 0.271, which are all much lower than 1.

If we instead define our recursive v_t equation as

$$v_t = \frac{\beta\, v_{t-1} + (1-\beta)\theta_t}{1 - \beta^t}, \tag{3.5.15}$$

then will scale up the first few instances of the data without affecting the latter entries, as shown in Figure 3.5.8. The term in the denominator of Equation 3.5.15 is known as the bias correction term for exponentially weighted averages.

In practice, most people do not actually calculate the bias correction because it does not make much of a difference on large datasets.

3.5.4 Gradient descent with momentum

In gradient descent, when we are far from an optimal point in our cost function, the gradients will be larger numbers. We are typically far at the beginning and taking large steps but often not in a great direction. In Figure 3.5.9, we are illustrating a potential path through the

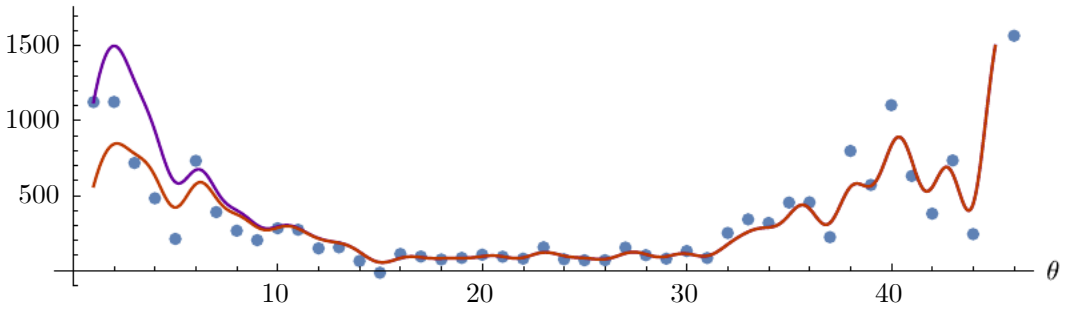


Figure 3.5.8: Weighted average of our data with bias correction (purple), without bias correction (red)

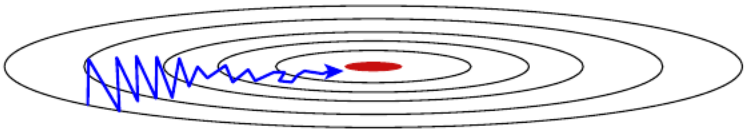


Figure 3.5.9: Contour plot of the cost function with the minimum in red and our path from gradient descent in blue

contour plot. Ideally, we would like to have a larger learning rate in the horizontal direction and a smaller learning rate in the vertical direction, in order to step more directly towards the minimum. Looking at the previous iterations of the direction of gradient descent, we can get a pretty good feeling about the direction of the minimum by taking the weighted average. In our figure, the vertical weighted average would be near 0, while the horizontal weighted average would be in the direction towards the center. Using the exponentially weighted average with gradient descent is referred to **gradient descent with momentum**. Gradient descent with momentum has two hyperparameters, α and β.

Algorithm 4 Gradient descent with momentum

$v_{dW}, v_{db} = 0$
repeat
  Compute dW, db with respect J
  $v_{dW} = \beta v_{dW} + (1-\beta)dW$
  $v_{db} = \beta v_{db} + (1-\beta)db$
  $W = W - \alpha\, v_{dW}$
  $b = b - \alpha\, v_{db}$
until the cost function J converges

Consider how a ball rolling down the sides of a bowl will find the minimum by taking a step in the direction of the gradient. Here, the equation

$$v_{dW} = \beta v_{dW} + (1-\beta)dW \tag{3.5.16}$$

has a loose analog to the physical system of a ball rolling down the sides of a bowl, where dW represents acceleration, v_{dW} represents velocity, and β represents friction.

Finally, some people may change the exponentially weighted average term to be

$$v_{dW} = \beta v_{dW} + dW, \tag{3.5.17}$$

where they omit $(1-\beta)$ in hopes of avoiding overfitting, however, this would require us to make less intuitive changes to our learning rate α in order for the cost function to converge quickly.

3.5.5 RMSprop

From Figure 3.5.10, notice that it would be more beneficial to move further in the direction with less oscillation than it would be to continue taking large steps vertically. One way to

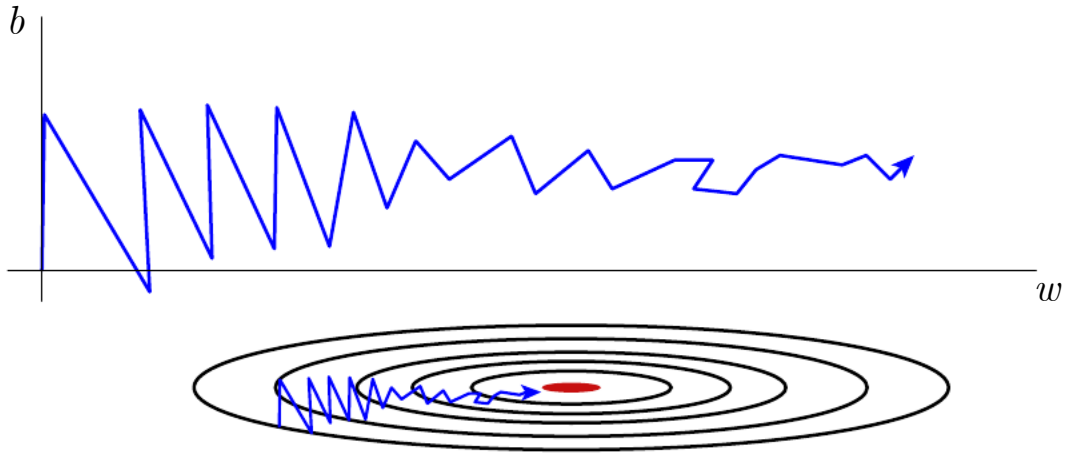


Figure 3.5.10: Plotting the gradient in each direction, if we denote the vertical axis as the bias and the horizontal direction with the weight

determine the amount of oscillation would be to calculate

$$\begin{cases} S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2 \\ S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2 \end{cases}, \tag{3.5.18}$$

where the square operation is element-wise and β_2 is new hyperparameter we are introducing that works similar to β.[2] Squaring the rate of change term will make the term positive and reward small values while making large values more costly. In our example, the changes in the horizontal direction, which correspond to the changes in the weights, are relatively small, so S_{dW} will be relatively small. In contrast, S_{db} will be huge because the vertical changes are massive. Now, if we update the weights and bias with

$$\begin{cases} w = w - \alpha \dfrac{dW}{\sqrt{S_{dW}} + \epsilon} \\ b = b - \alpha \dfrac{db}{\sqrt{S_{db}} + \epsilon} \end{cases}, \tag{3.5.19}$$

we will end up moving more in the direction that does not oscillate and less in the direction that oscillates a lot. Here, we include ϵ in order to avoid a division by zero error, where ϵ is commonly set to a small value such as $\epsilon = 10^{-8}$. The algorithmic described is known as **root mean square prop (RMSprop)** and is detailed in Algorithm 5

Algorithm 5 RMSprop

$S_{dW}, S_{db} = 0$
repeat
  Compute dW, db with respect J
  $S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2$
  $S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$
  $w = w - \alpha \dfrac{dW}{\sqrt{S_{dW}} + \epsilon}$
  $b = b - \alpha \dfrac{db}{\sqrt{S_{db}} + \epsilon}$
until the cost function J converges

[2]We are introducing the new hyperparameter primarily to make the next section which combines RMSprop and momentum together.

3.5.6 Adam optimization algorithm

The Adam optimization algorithm combines both RMSprop and momentum and has shown to work well across plenty of optimization problems. The complete algorithm is shown in Algorithm 6. Although we did not use bias correction with momentum or RMSprop, it is most common to use bias correction with the Adam algorithm.

Algorithm 6 Adam optimization

$S_{dW}, S_{db}, v_{dW}, v_{db} = 0$
repeat
$t = t + 1$
Compute dW, db with respect J
$v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) dW, \quad v_{db} = \beta_1 v_{db} + (1 - \beta_1) db$
$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2, \quad S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$
$v_{dW} = \frac{v_{dW}}{1 - \beta_1^t}, \quad v_{db} = \frac{v_{db}}{1 - \beta_1^t}$
$S_{dW} = \frac{S_{dW}}{1 - \beta_1^t}, \quad S_{db} = \frac{S_{db}}{1 - \beta_1^t}$
$w = w - \alpha \frac{v_{dW}}{\sqrt{S_{dW}} + \epsilon}$
$b = b - \alpha \frac{v_{db}}{\sqrt{S_{db}} + \epsilon}$
until the cost function J converges

From the Adam optimization paper (Kingma, et. al), the default values for the hyperparameters are

$$\beta_1 = 0.9 \tag{3.5.20}$$
$$\beta_2 = 0.999 \tag{3.5.21}$$
$$\epsilon = 10^{-8}. \tag{3.5.22}$$

The learning rate α does not have a default value and still needs to be tuned. **Adam** stands for **Adaptive moment estimation** and the terms β_1 and b_2 are the first and second moments, respectively.

3.5.7 Learning rate decay

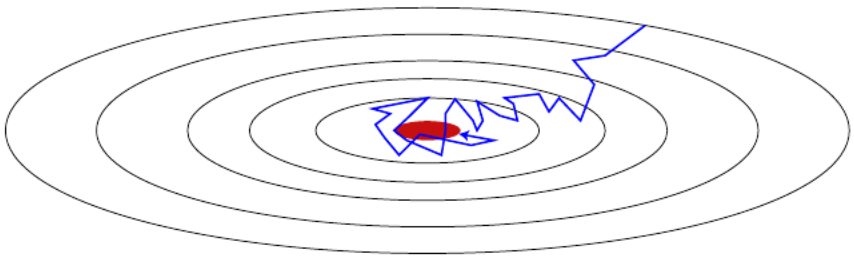


Figure 3.5.11: Gradient descent may take large steps when near the minimum, bouncing around and never quite converging

Learning rate decay refers to reducing the size of the learning rate as the number of iterations increases. Consider the example in Figure 3.5.11, where the steps once we are near the center appear to never quite reach the minimum and stay there. Instead, if we took smaller steps when closer to the minimum, we could more easily determine the values gradient descent is converging towards. One way to make α decay is by setting

$$\alpha = \frac{\alpha_0}{1 + decay\_rate * epoch\_num}, \tag{3.5.23}$$

where α_0 is the initial learning rate, *decay_rate* is a hyperparameter that we much set, and *epoch_num* is the number of times we have iterated through the entire training set. A few other ways to decay α include expotential decay

$$\alpha = \alpha_0(0.95^{epoch\_num}), \tag{3.5.24}$$

or

$$\alpha = \frac{k\,\alpha_0}{\sqrt{epoch\_num}}. \tag{3.5.25}$$

Occasionally, people may use a discrete staircase to change the learning rate, in which case the value changes after a set number of iterations, or we can go in and manually change the decay rate.

3.5.8 The problem with local optima

Back in the day, deep learning practitioners feared that their algorithms would find some local minimum and get trapped inside due to the gradient being 0 at a local optimum. In higher dimensional space, however, it is quite difficult for local optima to form because it would require every single direction to have the minimum at a single point. In general, we are more likely to see saddle points and plateaus. Plateaus tend to be the biggest problem because they will slow down the rate of gradient descent and could possibly fool us into believing that we have found a global minimum if we stop running gradient descent early.

3.6 Hyperparameter Tuning

3.6.1 Tuning process

With deep neural networks, there are quite a few hyperparameters that need to be tuned, including the learning rate, momentum term, number of layers, number of hidden units, learning rate decay, and mini-batch size. The learning rate tends to be the hyperparameter that can vary the most and is often the one that is hardest to tune. The next set of hyperparameters that can make a significant difference include changes to the momentum term, number of hidden units, and mini-batch size. The hyperparameters that represent the number of layers and learning rate decay may make a difference, but not to the degree of the rest of the hyperparameters mentioned.

When testing out choices for hyperparameters, it is often best to start with a reasonable range of values that could ponentially work for each hyperparameter. Then, rather than changing one hyperparameter at a time in a grid-like way, it is best to randomly choose values in the range for the hyperparameter to take on, which will allow us to test a lot more hyperparameters. For example, in Figure 3.6.1, randomly choosing hyperparameter 1 each time allows us to test 25 different values, while a grid choice would only allow for 5 different values we could test.

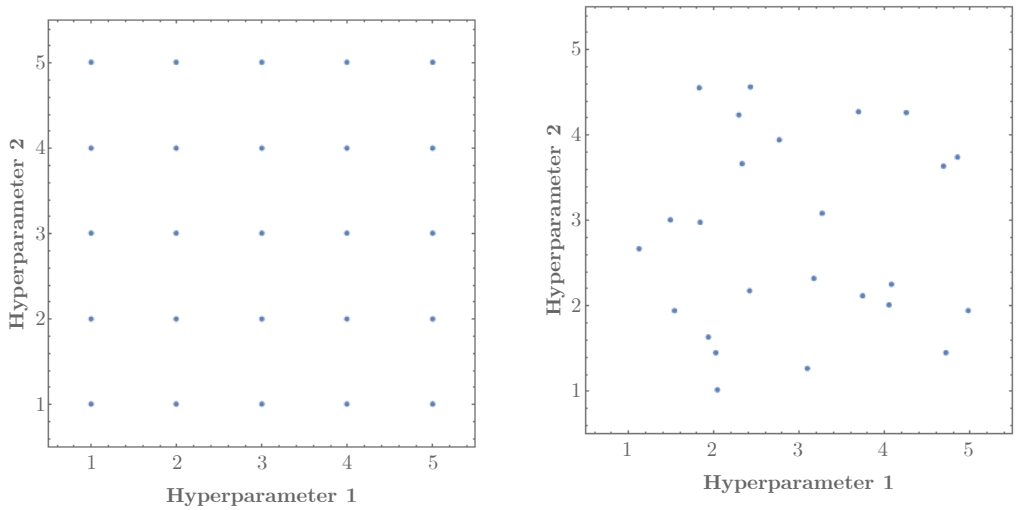
If we find that a section of the choices for our hyperparameters is producing good results, we can often zoom in to that section on focus more on the hyperparameters there, narrowing down the range of values. This is known as **coarse to fine** and is shown in Figure 3.6.2.

3.6.2 Using an appropriate scale to pick hyperparameters

Randomly testing hyperparameter values tends to work out well for things like the number of hidden units in each layer and the number of total layers. In each of these cases, we may only be choosing from around 50 different integer values. For the learning rate, however, the range might be from 0.0001 to 1 if we randomly choose values then 90% of the values we choose will be between 0.1 and 1. Instead of using a numerical scale, we may use a log scale to divide up the values.

To choose α from the logarithmic scale in Figure 3.6.3, we could first choose a random variable r that is in between 0 and -4, and then take

$$\alpha = 10^r. \tag{3.6.1}$$



(a) Grid choice of hyperparameters

(b) Random choice of hyperparameter values

Figure 3.6.1: Ways we can go about choosing different hyperparameters

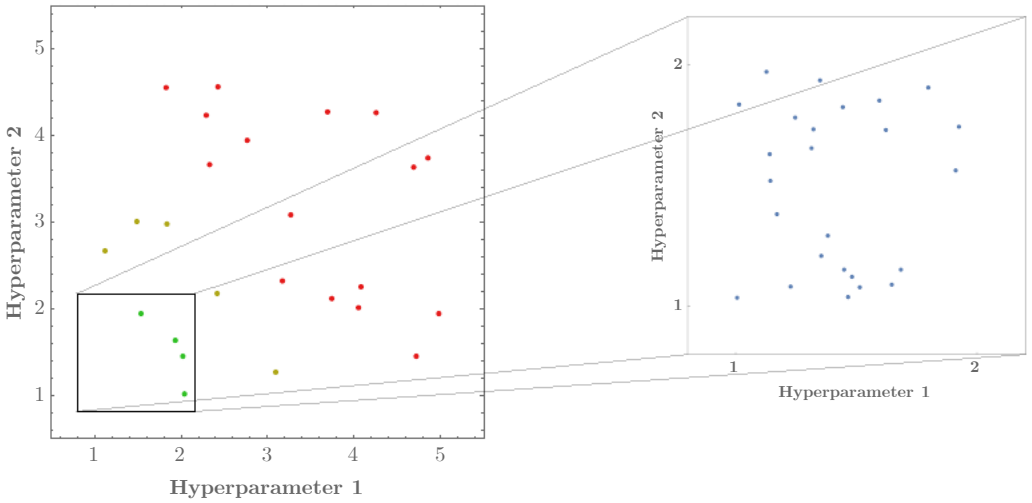


Figure 3.6.2: Coarse to fine, where we focus on the region of the choices for hyperparameters that has produced the most promising results.

Another problem that may arise is sampling the values of β, which are often between 0.9 and 0.999. If we instead look to find values of $1-\beta$, we will be searching for values between 0.1 and 0.001. Now, using the logarithmic scale just introduced, we can pick a random variable r between -3 and -1 and set

$$\beta = 1 - 10^r. \tag{3.6.2}$$

3.6.3 Hyperparameters tuning in practice: pandas vs caviar

For models that we do not have an immense GPU power to run on, we may be only able to train a single test over a certain amount of time. During this training period, we could proactively adjust the hyperparameters as the model is training, attempting to improve the process at each change. Here, we are taking a careful approach to make sure we get everything right on this model as best as we can, since we will only be testing a single model at a time. This is known as babysitting one model and is considered the **pandas approach** because pandas take careful care of their children. In contrast, if we multiple GPUs to run

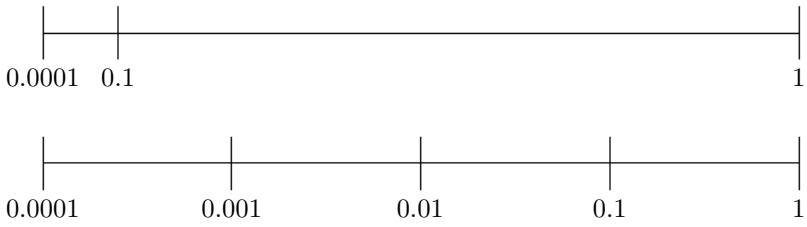


Figure 3.6.3: The bin sizes for randomly variables on a linear scale (top) versus a logarithmic scale (bottom)

our model across, we may run a bunch of tests in parallel and use the hyperparameters that give us the best results, which is known as the **caviar approach**.

3.7 Batch normalization

3.7.1 Normalizing activations in a network

Previously, we have been normalize our inputs $a^{[0]}$ to more easily train $w^{[1]}$ and $b^{[1]}$. Batch normalization takes this a step further and works to normalize $a^{[\ell]}$ in order to better train $w^{[\ell+1]}$ and $b^{[\ell+1]}$. In general, people are more likely to normalize $z^{[\ell]}$ instead of $a^{[\ell]}$, so that is what we will be focusing on.

To calculate batch normalization on layer ℓ, we will have the batch of values $\{z^{[\ell](1)}, \dots, z^{[\ell](m)}\}$, where

$$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{[\ell](i)} \tag{3.7.1}$$

and

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z^{[\ell](i)} - \mu\right)^2. \tag{3.7.2}$$

Just in case σ^2 turns out to be 0, we will introduce a hyperparameter ϵ and set the normalization of z to be

$$z^{[\ell](i)}_{norm} = \frac{z^{[\ell](i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{3.7.3}$$

where ϵ is some small number like 10^{-8}. We may not always want the mean to be 0 and the variance to be 1, so instead we will set

$$\tilde{z}^{[\ell](i)} = \gamma z^{[\ell](i)}_{norm} + \beta, \tag{3.7.4}$$

where both the new standard deviation γ and the new mean β can be learned from the model and are updated after each gradient descent iteration.

3.7.2 Fitting batch norm in a neural network

To add batch normalization to our model, each neuron will now have to calculate the linear activation, then the batch normalization of the linear activation, and then the non-linear activation. For each layer ℓ, the batch normalization step will have the parameters $\beta^{[\ell]}$ and $\gamma^{[\ell]}$ as shown in Figure 3.7.1. For the entire neural network, our parameters will be $W^{[\ell]}$, $b^{[\ell]}$, $\beta^{[\ell]}$, and $\gamma^{[\ell]}$ for $\ell \in \{1, \dots, L\}$, with our original gradient descent step updating to

$$\begin{cases} W^{[\ell]} = W^{[\ell]} - \alpha\, dW^{[\ell]} \\ b^{[\ell]} = b^{[\ell]} - \alpha\, db^{[\ell]} \\ \beta^{[\ell]} = \beta^{[\ell]} - \alpha\, d\beta^{[\ell]} \\ \gamma^{[\ell]} = \gamma^{[\ell]} - \alpha\, d\gamma^{[\ell]} \end{cases}. \tag{3.7.5}$$

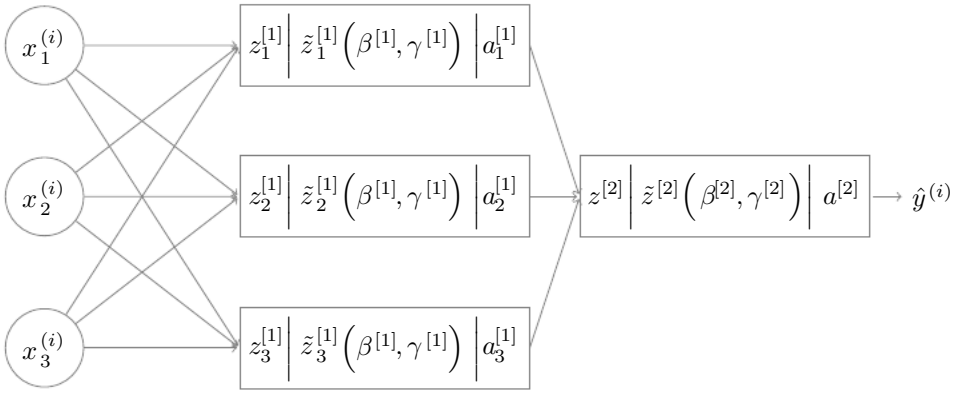


Figure 3.7.1: With batch normalization we now have three calculations at each node: the linear activation, the batch normalization of the linear activation, and the non-linear activation.

In our current process, we have

$$z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} \tag{3.7.6}$$

which is then normalied with

$$\tilde{z}^{[\ell]} = \gamma^{[\ell]} \left(\frac{z^{[\ell]} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta^{[\ell]} \tag{3.7.7}$$

$$= \gamma^{[\ell]} \left(\frac{W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta^{[\ell]} \tag{3.7.8}$$

$$= \gamma^{[\ell]} \left(\frac{W^{[\ell]} a^{[\ell-1]} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta^{[\ell]} + b^{[\ell]} \left(\frac{\gamma^{[\ell]}}{\sqrt{\sigma^2 + \epsilon}} \right). \tag{3.7.9}$$

Here, notice that bias term is not really adding any new information and we can combine the $\beta^{[\ell]}$ term with the $b^{[\ell]}$ term into a single term. For simplicity, we will choose to remove the term $b^{[\ell]}$ and keep $\beta^{[\ell]}$, giving us the updated parameters $\gamma^{[\ell]}$, $W^{[\ell]}$, and $\beta^{[\ell]}$.

Currently, we have been using batch normalization with the entire training set, but if we were to break the data up into mini-batches, we would go through a similar process using batch normalization on each mini-batch. The complete algorithm for gradient descent with batch normalization is described in Algorithm 7

Algorithm 7 Gradient descent with batch normalization

for $t = 1$ to the number of mini-batches **do**
  Forward propagate on $X^{\{t\}}$
    In each hidden layer update $z^{[\ell]}$ with batch normalization
  Backward propagate to compute $dW^{[\ell]}$, $d\beta^{[\ell]}$, $d\gamma^{[\ell]}$
  Update the parameters with $W^{[\ell]} = W^{[\ell]} - \alpha\, dW^{[\ell]}, \ldots$
end for

3.7.3 Batch norm at test time

Recall that for each mini-batch, we calculated

$$\begin{cases} \mu = \dfrac{1}{m}\displaystyle\sum_{i=1}^{m} z^{(i)} \\ \sigma^2 = \dfrac{1}{m}\displaystyle\sum_{i=1}^{m}\left(z^{(i)} - \mu\right)^2 \\ z_{norm}^{(i)} = \dfrac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta \end{cases}, \tag{3.7.10}$$

which relies on having an entire training set of data, which we do not have at test time. In practice, we predict μ and σ^2 as the exponentially weighted average across mini-batches. For example, if we had three mini-batches, then for each layer ℓ we would find $\mu^{\{1\}[\ell]}$, $\mu^{\{2\}[\ell]}$, $\mu^{\{3\}[\ell]}$ and $\sigma^{2\{1\}[\ell]}$, $\sigma^{2\{2\}[\ell]}$, $\sigma^{2\{3\}[\ell]}$ and calculate the exponentially weighted average for both μ and σ^2. At test time, we would compute

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{3.7.11}$$

where μ and σ^2 come from the latest calculation of our exponentially weighted average.

3.8 Multi-class classification

3.8.1 Softmax regression

Currently, we have only been dealing with binary classification problems, but it would be much more useful to be able to handle multi-class classifications.

Figure 3.8.1: Multi-class classification with the labels are 0 → cat, 1 → dog, 2 → lion, and 3 → zebra.

We will denote C to be the number of classes, so in Figure 3.8.1 $C = 4$. The number of classes will correspond to the number of neurons in the last layer of the neural network, so

$$n^{[L]} = C, \tag{3.8.1}$$

where each neuron corresponds to the probability of each class as shown in Figure 3.8.2.

From the laws of probability, we know that the sum of the final layer must be 1. We will start off, as usual, calculating the linear activation in the final layer, but we must change the activation function. For our activation function, let us first calculate the element-wise exponentiation of each linear activation

$$t = e^{\left(z^{[L]}\right)}, \tag{3.8.2}$$

where t is a temporary variable and $t \in \mathbb{R}^{C\times 1}$. Next, we will normalize the vector t which will be our activation function

$$a^{[L]} = \frac{e^{\left(z^{[L]}\right)}}{\sum_{j=1}^{C} t_j}, \tag{3.8.3}$$

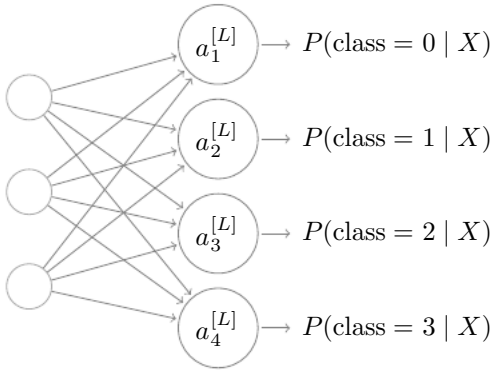


Figure 3.8.2: Each neuron in the final layer corresponds to a probability of a class

where $a^{[L]} \in \mathbb{R}^{C\times 1}$. Equation 3.8.3 is again an element-wise function where the ith element is

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{C} t_j}. \tag{3.8.4}$$

As a concrete example, suppose we have

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}. \tag{3.8.5}$$

We will first exponentiate each element

$$t = e^{\left(z^{[L]}\right)} = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \approx \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \tag{3.8.6}$$

and calculate

$$\sum_{j=1}^{4} t_j \approx 176.3. \tag{3.8.7}$$

Next, we will divide each element in t by the sum of all the values in t, which gives us the probabilities each class occurs

$$a^{[L]} = \begin{bmatrix} e^5/176.3 \\ e^2/176.3 \\ e^{-1}/176.3 \\ e^3/176.3 \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}. \tag{3.8.8}$$

The activation function we have been using on the final layer of our neural network is known as the **softmax activation function**. The name softmax is in contrast to a **hardmax classifier** that would output 1 for the max and 0 for everything else. In our example, the hardmax classifier would output

$$a^{[L]} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.8.9}$$

3.8.2 Training a softmax classifier

As an example, suppose our neural network outputs the data

$$\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \tag{3.8.10}$$

where the actual labels are

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} . \tag{3.8.11}$$

The loss function we will use for a softmax classifier is defined as

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j \log \hat{y}_j, \tag{3.8.12}$$

which would leave us with only the prediction from the positive label contributing to the loss and we can simplify our loss to be

$$\mathcal{L}(\hat{y}, y) = -y_2 \log \hat{y}_2 = -\log \hat{y}_2. \tag{3.8.13}$$

In order to minimize our loss, we would want to make $\hat{y}_2$ as close to 1 as possible. If $\hat{y}_2$ is near 0^+, then we will have a loss approaching ∞. For a multi-class classifier, we will use the same definition of the cost function from before with

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right). \tag{3.8.14}$$

We can calculate the backward propagation step through a softmax classifier with

$$\frac{\partial J}{\partial z^{[L]}} = \hat{y} - y. \tag{3.8.15}$$

Chapter 4

Applied deep learning

4.1 Orthogonalization

We will use the term **orthogonalization** to refer to changes in one hyperparameter not influencing the rest of the hyperparameter. For instance, in a self-driving car, we would not want changes in the acceleration to have any effect on the steering. In regards to machine learning, we primarily have to fit the training set well, fit the dev set well, fit the test set well, and then maintain the model on real-world examples. If one of these steps is not working well, then we would like to adjust a set of hyperparameters that will have little impact on the other steps.

4.2 Setting up goals

4.2.1 Single number evaluation metric

Using a binary cat classification example, we will say that **precision** refers to the percentage of actual cats inside of the recognized cats by our model. **Recall** refers to the percentage of actual cats that are correctly labeled.

Classifier	Precision	Recall
A	95%	90%
B	98%	85%

Table 4.1: Example of precision and recall for two different classifiers

Suppose our classifier gives us the precision and recall data in Table 4.1, where one classifier gives us better precision, while the other gives us better recall. With two evaluation metrics, determining the best classifier is hard to do. Instead, we can combine precision and recall into a category known as **F1 Score**, where

$$\text{F1 Score} = \frac{2}{\frac{1}{P} + \frac{1}{R}}, \tag{4.2.1}$$

where P represents precision, and R represents recall. This way of calculating the mean is known as the **Harmonic mean**. Now, using the F1 Score, we can simply say that classifier A is the better classifier.

When choosing metrics, we typically first want to define a metric and then have a completely separate process the deals with optimizing that metric. It is often best to quickly choose an evaluation metric and then change that metric in the future if there is a significant difference in how data performs on the development set versus the testing set versus the real world.

Classifier	Precision	Recall	F1 Score
A	95%	90%	92.4%
B	98%	85%	91.0%

Table 4.2: Calculating the F1 Score for each classifier

4.2.2 Train/dev/test distributions

For our development and test set, we would like to have instances come from the same distribution of our data. In particular, we would like each of these sets to model the expected data we will receive in the future as best as possible. The size of each set may vary. Our test set should be large enough that it is capable of giving us a lot of confidence if our model performs well on it.

4.3 Comparing to human-level performance

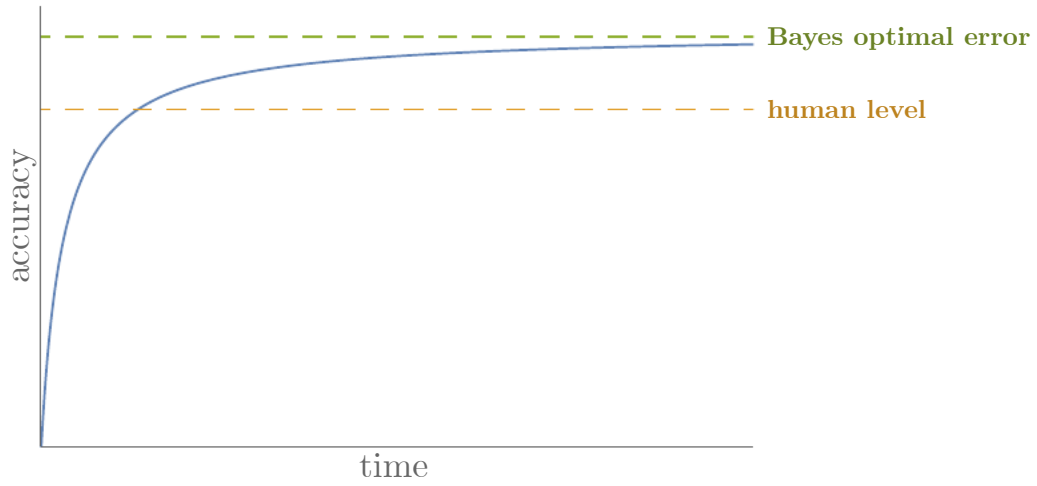


Figure 4.3.1: How our model could behave over time

In cases such as image recognition or speech detection, we tend to be able to create models that rapidly approach human-level performance, but then slowly decrease the rate at which the accuracy is improving. Some of the reasons for the plateau after human-level performance is achieved are that human-level and Bayes optimal error are not too far apart on tasks, like recognizing images. When our model performs worse than human-level performance we still have ways we can improve with gaining more trained data and manual error analysis. Our model will never surpass the **Bayes optimal error**, which is the *best error rate* we could possibly achieve as shown in Figure 4.3.1. It will not always be 100%, because in scenarios where images are blurred, for example, it may be impossible to extract information about what is in the image.

4.3.1 Avoidable bias

We will hardly ever actually have a number for Bayes optimal error. Instead, we use human-level error to approximate Bayes optimal error. From Table 4.3, we have two different tasks that give us the same training and development error but differ in human error. In task A, where our training error is quite far from our human error, we likely want to focus on improving the bias in our model. In contrast, when our training error is close to our human error, like in model B, it may be best to focus on improving variance in the model. We say that the difference between our approximation for Bayes optimal error and our training

Task	A	B
Human Error	1%	7.5%
Training Error	8%	8%
Development Error	10%	10%

Table 4.3: Example error rates for two tasks with different human error rates.

error is the **avoidable bias** and the difference between the training error and development error is the **variance**.

4.3.2 Understanding human-level performance

Suppose we have the medical imaging classification error rates:

- Typical human: 3%
- Typical doctor: 1%
- Experienced doctor: 0.7%
- Team of experienced doctors: 0.5%

and we want to determine a human-level error. Since human-level is often used as an approximate for Bayes optimal error, it is best, in this case, to go with the error rate of 0.5% from the team of experienced doctors. For deployment purposes, however, we may have an application that just needs to beat a typical doctor, in which case it may be best to use the typical doctor's error rate of 1% as human-level.

4.3.3 Surpassing human-level performance

If we have the error rates

- Team of humans: 0.5%
- One human: 1%
- Training error: 0.3%
- Dev error: 0.4%

where we have already surpassed our estimate for Bayes optimal error, then it is hard to determine what we should focus on improving next. There are a bunch of problems that fall into categories where machine learning far outperforms human-level performance such as online advertising, product recommendations, transit time predictions, and loan approvals.

4.4 Error analysis

Suppose that we are working on a cat classifier and we have achieved a 10% error rate, with a Bayes optimal error rate of 2%. Further, suppose we also want to see if better recognition of dogs could help improve our cat classifier if it mislabels a dog as a cat. Before attempting for our cat recognizer to do better with dogs, it is likely best to find out how many images are mislabeled in our development set and how many of those images are dogs. If we have around 100 mislabeled images in total and only 5 of those images contain dogs, then the **ceiling** we can improve our model at by recognizing dogs is with a 9.5% error rate. Instead, if we 50 images where dogs are mislabeled in our development set, then improving dog classification would be more worth our time, since the ceiling for our development set error rate would be 5%.

We can also look for multiple sets of misclassified images in parallel. For example, on our cat classification, we may want to check for dogs, great cats (lions, panthers), and blurry images being mislabeled. To check for multiple categories, it may be best to create a table or spreadsheet like the one shown in Table 4.4.

Image	Dog	Great cats	Blurry	Comments
1	✓			Pitbull
2			✓	
3		✓	✓	Rainy day at zoo
⋮		⋮		
Total %	8%	43%	61%	

Table 4.4: Using a table to represent our misclassified images

4.4.1 Cleaning up incorrectly labeled data

With a large enough training set, it may not matter much if data is randomly labeled incorrectly. However, for smaller data sets or if data is systematically labeled incorrectly it may be in our best interest to manually change the labels. In our development set, we may update our table to include a column for incorrectly labeled instances as shown in Table 4.5.

Image	Dog	Great cats	Blurry	Incorrectly labeled	Comments
⋮					
98				✓	Cat was in background
99		✓			
100				✓	Cat drawing
Total %	8%	43%	61%	6%	

Table 4.5: Adding an incorrectly labeled category to our table of misclassified images

We tend to only make adjustments to the labels in our development set when the changes could produce significantly better results. For example, if our development set error was 10% and the error due to incorrect labels was 0.6%, then it may be the most worthwhile to focus on the 9.4% first. If we later get the development set error rate down to 2%, while still having 0.6% error due to incorrect labels, then it may be worthwhile to adjust the labels so that they are accurate. If we go in and adjust the development set for errors in the mislabeled images, then it may also be worthwhile to go in and look for incorrectly labeled images that our network labeled as correct.

4.5 Mismatched training and dev set

Figure 4.5.1: Training data could vary significantly from data our model views once deployed

After developing a model, we may test it in the real world and find that the data uploaded by users varies significantly from the data we trained our model on. Consider the data in Figure 4.5.1, which shows that we have 200,000 images from our training set and 10,000

images from the users that each is distributed differently. We also know that we want to optimize the uploaded data from the users as best as possible, although 10,000 images will not work well as an entire training set.

One way we could adjust our sets would be to combine the testing and uploaded data into a single large array, then shuffle the array and break up the data into groups for a training, development, and testing set. Suppose we set our training set size to be 205,000 and our development set size and testing set size each to 2,500. If we were to break data up randomly, then we would only expect around 119 images in the development and test set to come from the uploaded data from users.

Instead, since our primary goal is to optimize the data uploaded by users, we could again use bin sizes of 205,000 for the training set and 2,500 for both the development and testing set, but this time we will make up the development and testing sets entirely from the uploaded data from users. While the data is not split evenly, we tend to prefer this distribution of data because it allows us to focus more on optimizing the right goals.

4.5.1 Bias and variance with mismatched data distributions

When we distribute the types of data differently between our training set and development set, it may be likely that we have a model with a low training error and a relatively high development error, even without having a variance problem. Instead of directly comparing the results against the development set, we will introduce a **training-development set**, which has the same distribution as the training set although we do not actually use data in this set to train on. Now, we can look at the training set error rate, the training-development set error rate, and the development set error rate, where we will determine the variance based on the difference between the training set error rate and the training-development set error rate. If we have both a low training set error rate and training-development set error rate while our development set error rate is high, then we say we have a **data mismatch** problem.

4.5.2 Addressing data mismatch

With a data mismatch problem, it may first be beneficial to look at where the errors are coming from in the development set. If we find, for example, that our errors are due to blurry images then we may want to get more training examples of blurry images. Instead of actually going out and getting more data, we could use **artificial data synthesis**, which in this case would take crisp images and apply a blurring filter to them.

For another example, if we are trying to create a trigger word detector and we have a bunch of low noise voice recordings, but we find that the users' data once our model is deployed tends to have a lot of noise, then we can artificially add car noise. One thing to keep in mind here is that if we have a lot more data of voice recordings than we have of car noise, adding the same car noise to each audio clip may cause our model to overfit the car noise data.

4.6 Learning from multiple tasks

4.6.1 Transfer learning

Transfer learning is a process of taking a neural network that has been trained for one thing and using it as a starting point to train something else. For instance, if we are training an image classification system, we may be able to transfer the low-level implementation details in a neural network, that ideally look for edges and curves, and use those details with radiology diagnosis.

When changing the last layer of a neural network as shown in Figure 4.6.1, we should set the new weight and bias terms randomly. We are also able to add more layers to a network if necessary and are not restricted to simply changing out the last layer when using transfer learning.

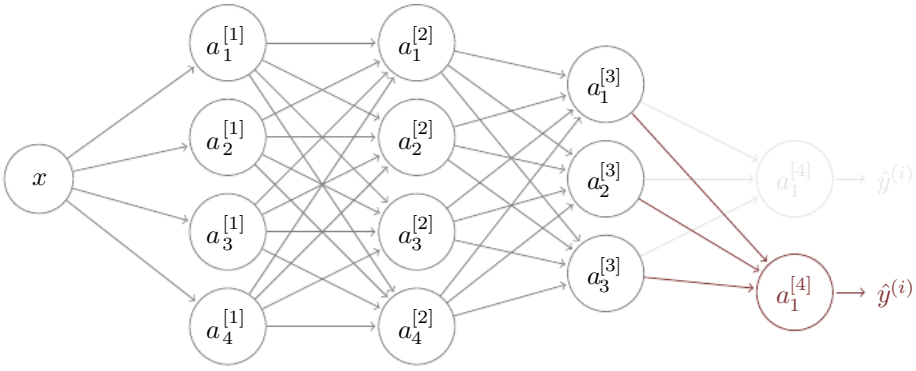


Figure 4.6.1: Disconnecting the end of a neural network and *transfering* the trained each weight and bias to another similar problem

In our image classification to radiology diagnosis example, we say that the image classification data is the **pre-training** data, while the radiology diagnosis data is the **fine-tuning** our model.

We will often change the number of layers that are learned in a neural network based on the amount of fine-tuning data we have available. With more fine-tuning data it would be beneficial to use more layers, while with only a little bit of fine-tuning data we would rely more heavily on our pre-training data. It is also common to freeze the representations of earlier layers in a neural network and only learn the disconnected part, or the latter part of the model. With plenty of data, we may use the pre-training model as our initialized representations and retrain the entire network.

4.6.2 Multi-task learning

For multi-task learning, suppose that we are building a self-driving car and need to classify pedestrians, cars, stop signs, and traffic lights. Instead of building four separate neural networks to do each of these tasks, we can use a single neural network with multi-hot encoding where our outputs are labeled with

$$y^{(i)} = \begin{bmatrix} \text{pedestrian?} \\ \text{car?} \\ \text{stop signs?} \\ \text{traffic light?} \end{bmatrix} \tag{4.6.1}$$

and $a^{[L]} = 4$. If some of our labels do not have all the information filled in, for instance if it is unknown if there is a stop sign or traffic light in the image, then we can have our loss function not sum over those values. Multi-task learning is different from softmax regression because each image can have multiple labels.

Multi-task learning works best with tasks that share low-level features, relatively similar data, and when we can use a large enough neural network.

Chapter 5

Convolutional neural networks

With our current approach to image classification, where we took pixels as our input, we will quickly find that with larger images or images with different aspect ratios, our neural network will no longer work. Consider an image of size $1000 \times 1000 \times 3$ which would make our input of size 3 million. With such a large input, it is too computationally expensive to run most models and overfitting is prone to occur. Instead, we will use and introduce convolutional neural networks in this section.

5.1 Edge detection

From the field of image processing, we have long been able to apply **filters** to images. We represent different filters as matrices or tensors. Applying a filter to a set of pixels can give us properties of an image, including where the edges are. When we apply a filter to a matrix, the matrix and the filter must be the same size and the result is the element-wise sum between the filter and matrix.

With images, the matrix that represents the pixels is usually larger than the filter. In cases where our matrix is of a different size than the filter, we can apply a **convolution** operation that will result in a matrix all the possible ways to form complete filters across the image. It is typical to denote the convolution operator as $*$, which is not to be confused with the element-wise operator we defined with the same notation earlier.

We will denote filters to be of size $f \times f$ and our image matrix of size $n \times n$. After convolving a filter across our image matrix, we will end up with a matrix of size $(n-f+1) \times (n-f+1)$. An example of a convolution operator can be found in Figure 5.1.1.

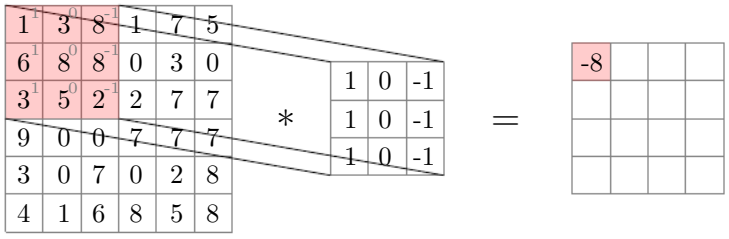


Figure 5.1.1: A convolution over a matrix is the element-wise product between a filter and smaller entries of the matrix. The value from the convolution between the filter and the upper left-hand corner of the matrix corresponds to the top left entry in the result of the convolution.

With different filters, we can find out different properties from our image. When looking at results after applying an edge detection filter, gray represents no edge detected, black represents positive values from our filter, and white represents negative values from our filter, as shown in Figure 5.1.2. Some of the most common filters used for edge detection are shown in Figure 5.1.3.

(a) Original image **(b)** Horizontal edge detector **(c)** Vertical edge detector

Figure 5.1.2: Resulting images after applying a horizontal and vertical filter

1	0	-1
1	0	-1
1	0	-1

(a) Standard vertical filter

1	0	-1
2	0	-2
1	0	-1

(b) Sobel vertical filter

3	0	-3
10	0	-10
3	0	-3

(c) Scharr vertical filter

1	1	1
0	0	0
-1	-1	-1

(d) Standard horizontal filter

1	2	1
0	0	0
-1	-2	-1

(e) Sobel horizontal filter

3	10	3
0	0	0
-3	-10	-3

(f) Scharr horizontal filter

Figure 5.1.3: Different types of horizontal and vertical filters that are used for edge detection

5.2 Padding

If we stack multiple convolution operations back to back, then we will end up with a significantly smaller final outputted image than we started with. One way to combat the changing size would be by adding padding to the image. When we add padding to the image, we typically add a border of all 0s as shown in Figure 5.2.1.

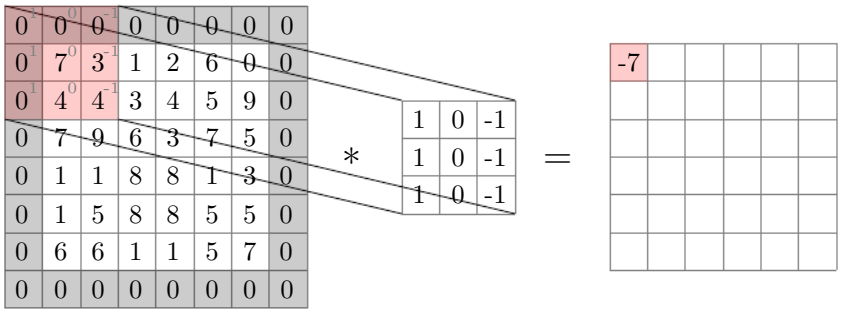


Figure 5.2.1: Adding padding to our image could produce the output in the same size as our input

We add padding uniformly to each size of the image and the amount of padding we add is denoted by p. With padding, the output of our image will be of size $(n + 2p - f + 1) \times (n + 2p - f + 1)$. If we want our input to be equal to our output, then we can solve $n + 2p - f + 1 = n$ for p and find

$$p = \frac{f-1}{2}. \tag{5.2.1}$$

When the output size matches the input size, we call the convolution a **same convolution** and when we do not add padding, we say that we have a **valid convolution**. It is also most common to see f as an odd number because that allows for a pixel to be in the distinctive center.

5.3 Strided convolutions

Figure 5.3.1: An invalid filter since we will not use the filters that run over the side of the image, instead we will only use filters that fill a complete $f \times f$ grid

With convolutions, we are also able to take stride lengths that are larger than 1. From the top left corner, when moving right or down we will move each spot in the filter a length of s. With strides, only count the convolutions that contain lie inside of the boundary of our image with padding, as shown in Figure 5.3.1. Therefore, the size of our output will be

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor. \tag{5.3.1}$$

5.4 Cross-correlation vs convolution

In image processing, the convolution operator actually mirrors the filter horizontally and vertically as shown in Figure 5.4.1. The method we have been using thus far, where we do not mirror the filter is known in image processing as cross-correlation. However, it is most common in computer vision to use the word convolution in place of cross-correlation, which is what we will continue to do.

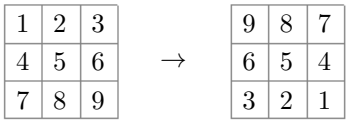


Figure 5.4.1: With a traditional convolution operation, we first flip the filter horizontally and vertically

5.5 Convolutions over volume

For colored images, we have three channels–red, green, and blue–and can define a filter for each channel as shown in Figure 5.5.1. For a channel with a height and width of f, our filter will have f^3 parameters. To convolve with volume, we apply the front filter only to the red channel, the second filter only to the green channel, and the third filter only to the blue channel. Then, on each channel, we carry out the convolution process described earlier.

Previously, we saw that each filter may be looking for different things inside of an image. We can also set up multiple filters, by storing the resulting matrices from each convolution inside of a tensor. We will denote the number of filters as n_c', so our outputted matrix will be n_c' layers deep as shown in Figure 5.5.2.

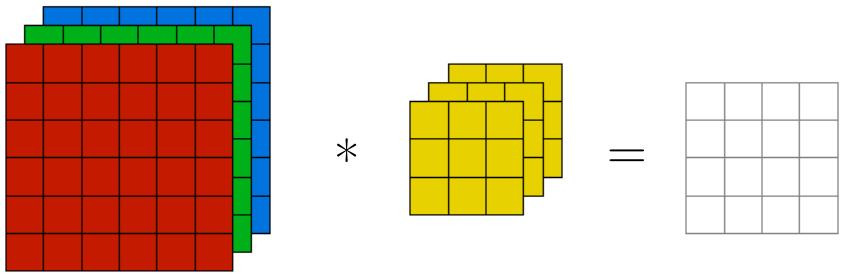


Figure 5.5.1: Convolutions of an RGB image

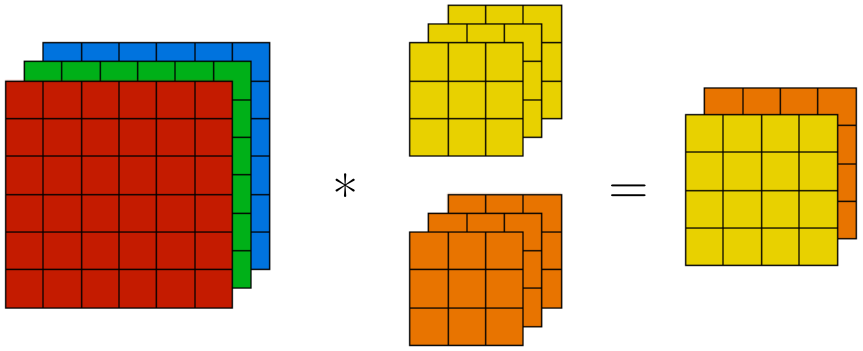


Figure 5.5.2: Using a convolution on n_c' filters will produce a matrix in n_c' dimensions with each dimension corresponding to a result from a convoluted filter

5.6 One layer convolution network

For convolutional neural networks, we will treat the resulting matrices created from the convolution added to the bias term b as our linear activation z. Then we will put each element through a non-linear activation as shown in Figure 5.6.1. Each index inside of the filter tensor is treated as a weight.

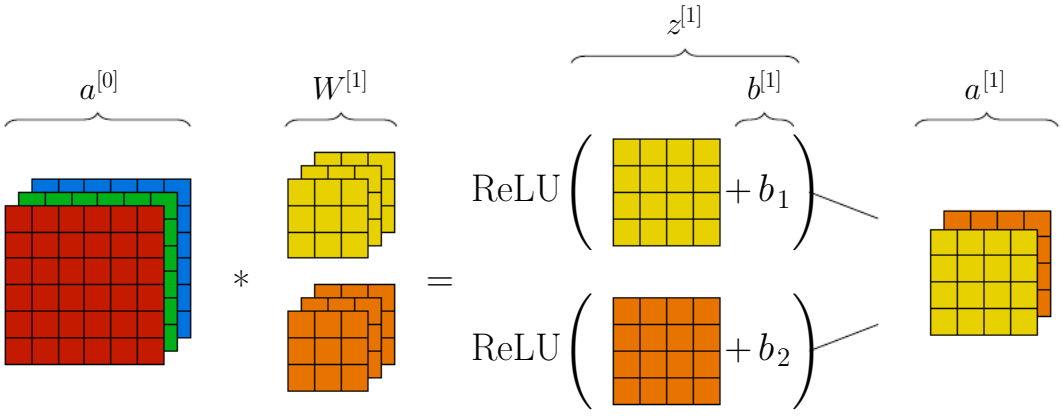


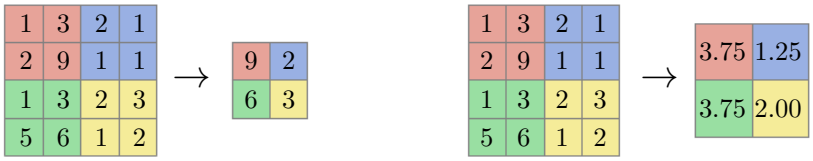
Figure 5.6.1: One layer of a convolutional neural network

For the lth layer of a convolutional neural network:

- $f^{[l]}$ = filter size
- $p^{[l]}$ = padding
- $s^{[l]}$ = stride
- $n_c^{[l]}$ = number of filters
- $n_h^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$ = size of input, height of input × width of input × channels in input
- $n_h^{[l]} = \left\lfloor \frac{n_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$
- $n_w^{[l]} = \left\lfloor \frac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$
- $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$ = filter size
- $n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]} = a^{[l]}$
- $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ = size of weights
- $n_c^{[l]}$ = size of bias

5.7 Pooling layers

Pooling layers give us a way to shrink the size of our input, while still capturing the essential details. The primary types of pooling include average pooling and max pooling. Pooling layers have both a filter and a step size, although there are no weights to be learned. Instead, pooling applies a function to each region of the input it focuses on, where each frame is $f \times f$ and the stride length is s. For 2D max pooling, the output is the maximum number inside of the frame and for 2D average pooling, the output is the average of all the numbers inside of the frame.



(a) Max pooling **(b)** Average pooling

Figure 5.7.1: 2D pooling with the hyperparameters $f = 2$ and $s = 2$

For 3D pooling, the number of channels in the input will match the number of channels in the output. Each of the channels will work independently. For the ith channel in the output, we will take the pool from the filter over the ith channel of the input.

Max pooling is used most commonly because it generally produces better results. Some people hypothesize that max pooling performs better because it captures in each filter if a certain aspect of the image is present, while average pooling can be saturated with a lot of data.

In deep learning literature, we combine convolution and pooling steps into a single layer because pooling does not have any representations to be learned.

5.8 Why we use convolutions

As a thought experiment, consider a single layer neural network with an input size of $32 \times 32 \times 3$ and an output size of $28 \times 28 \times 6$. If we were to use a fully-connected neural network, we would have around 14 million representations we would have to learn, whereas

if we use a convolutional neural network with a filter size of 5×5 then we would only need to learn 156 representations.

With fewer representations we can take advantage of **parameter sharing** and **sparsity of connections**. Parameter sharing allows us to detect features at any area inside of the image. The sparsity of connections means that for every output, there will only be a few numbers that impact it.

5.9 Classic networks

Since choosing the architecture for a neural network is often an empirical task we can look at other people's models to make better decisions on our model. The main way to look at how other people build their models is by reading their research. We will be going over how LeNet-5, AlexNet, VGG-16, ResNet, and Inception work.

5.9.1 LeNet-5

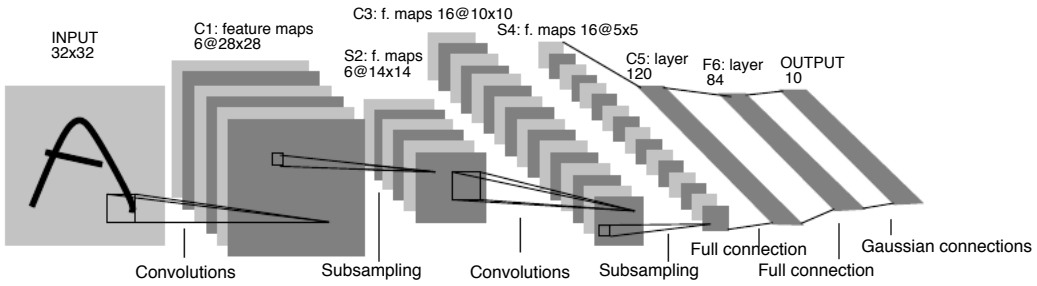


Figure 5.9.1: LeNet-5 architecture (LeCun et al., 1998).

LeNet-5, named after the paper's lead author Yann LeCun, was introduced in 1998 with a primary goal of recognizing characters. LeNet-5 uses average pooling which was more widely used at the time of publication. This model contains around 60 thousand representations, which is now considered a small amount. Among other things, this architecture coined the concept of moving deeper into the neural network to shrink the input size, while increasing the number of channels and convolutional layers should be followed by pooling layers.

5.9.2 AlexNet

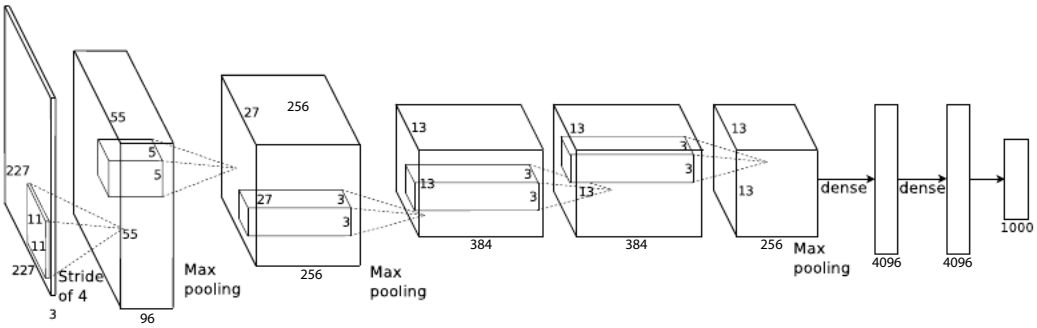


Figure 5.9.2: AlexNet architecture (Krizhevsky et al., 2012).

AlexNet looks quite similar to LeNet-5. Although, AlexNet has around 60 million parameters in total and used a ReLU activation function. AlexNet significantly sparked a surge of research toward artificial intelligence when published in 2012, because the results from the neural network far outperformed state-of-the-art approaches at the time.

5.9.3 VGG-16

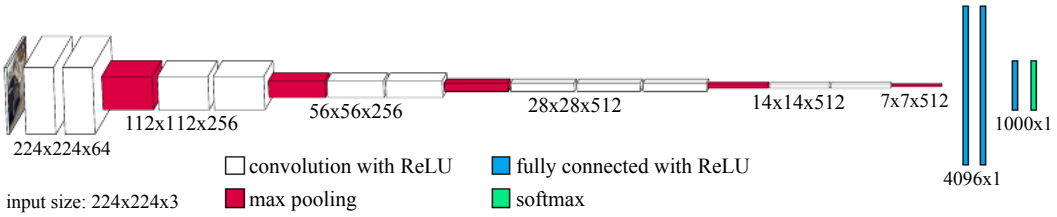


Figure 5.9.3: VGG-16 architecture

The VGG-16 architecture was introduced in a 2015 paper by Karen Simonyan and Andrew Zisserman. In it, they make all their convolutions as same convolutions with a filter size of 3×3 and a stride length of 1. Each max pooling layer has a filter size of 2×2 with a stride length of 2. This architecture is on the larger side of modern neural networks with a total number of representations around 138 million and sticks out primarily due to its simple choices for hyperparameters that turn out to work well.

5.9.4 ResNet

A residual network (ResNet) gives us a way to **skip connections** inside of a neural network. With a plain network, our path from $a^{[\ell]}$ to $a^{[\ell+2]}$ may be what is shown in Figure 5.9.4.

$$a^{[\ell]} \longrightarrow \text{linear} \longrightarrow \text{ReLU} \longrightarrow a^{[\ell+1]} \longrightarrow \text{linear} \longrightarrow \text{ReLU} \longrightarrow a^{[\ell+2]}$$

Figure 5.9.4: Plain network

However, with a ResNet, we use what is known as a residual block that sets

$$a^{[\ell+2]} = g\left(z^{[\ell+2]} + a^{[\ell]}\right), \tag{5.9.1}$$

where $a^{[\ell]}$ is skipping connections. The path of a residual block is shown in Figure 5.9.5.

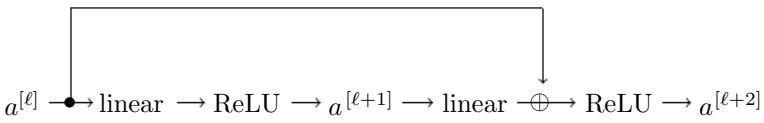


Figure 5.9.5: Residual block

As shown in Figure 5.9.6, residual networks work well with deep neural networks that have a lot of layers whereas plain neural networks often have weight decay.

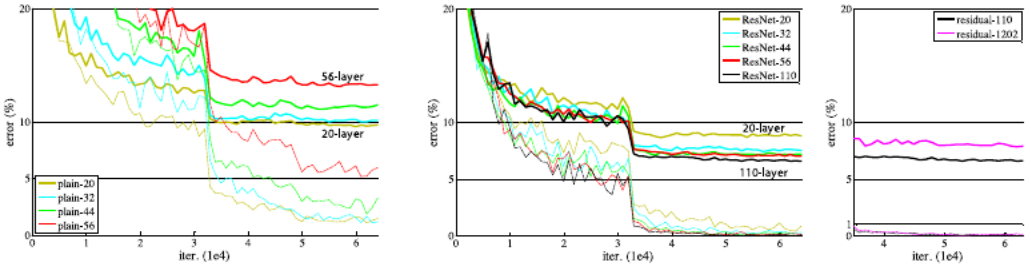


Figure 5.9.6: The error rate for plain networks **(left)** and ResNets **(middle, right)** when the number of layers in the network changes (He et al., 2015).

The reasoning why ResNets work with more layers can be shown by expanding the right-hand side of Equation 5.9.1

$$a^{[\ell+2]} = g\left(z^{[\ell+2]} + a^{[\ell]}\right) \tag{5.9.2}$$

$$= g\left(w^{[\ell+2]}a^{[\ell+1]} + b^{[\ell+2]} + a^{[\ell]}\right). \tag{5.9.3}$$

Notice that when weight decay occurs in plain neural networks, where $w^{[\ell+2]} \approx 0$, and we are using a ReLU activation function, we will come close to having $a^{[\ell+2]} = g\left(a^{[\ell]}\right) = a^{[\ell]}$. Therefore, adding more layers to a ResNet may have no effect on the activation if weight decay occurs, or the weights could add additional information that could decrease the error rate.

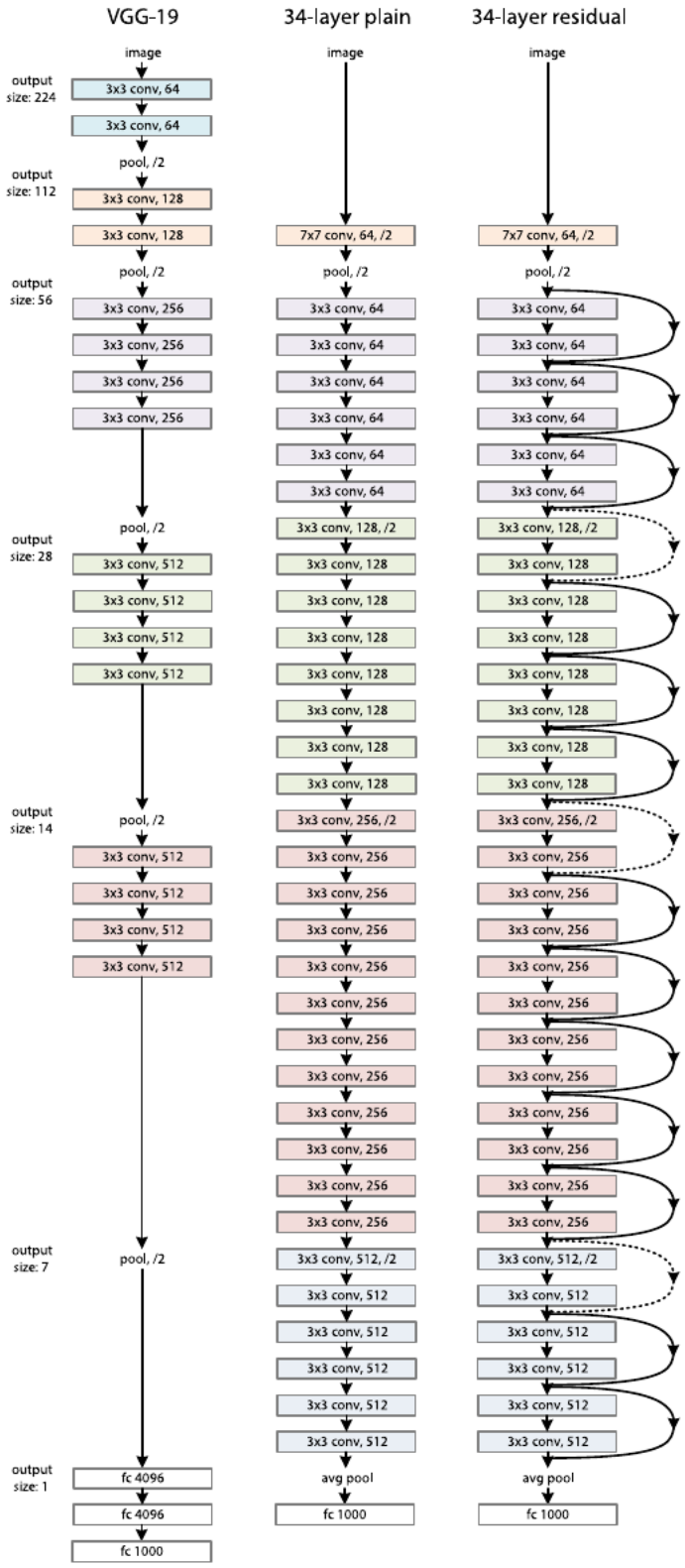


Figure 5.9.7: How a 34-layer ResNet compares to a 34-layer plain network and VGG-19, which is modified version of VGG-16 (He et al., 2015).

5.9.5 1×1 convolution

While a 1×1 convolution would simply scale up each value in a 2D input, with 3D inputs we can do much more. For a 1×1 convolution the depth of the filter will be equal to the depth of the input. Then the output for each filter comes from the dot product between the pixel on the input and the filter. Also, our output will always be the same size as our input. With multiple filters, we add depth to the output as shown in Figure 5.9.8. 1×1 convolutions may also be referred to as a network in network (Lin et al., 2013).

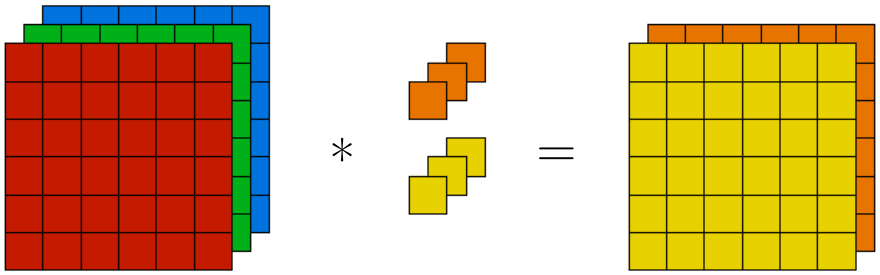


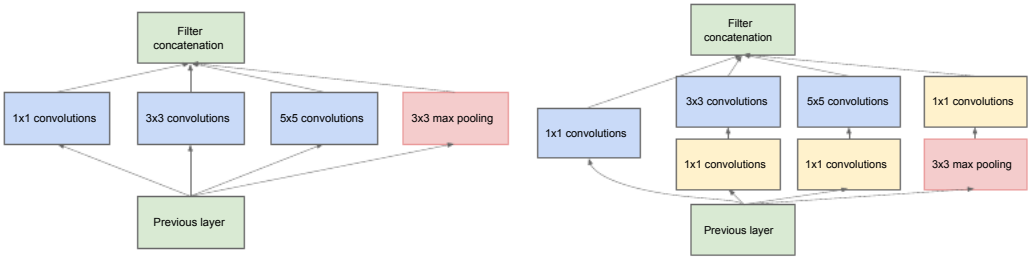
Figure 5.9.8: 1×1 convolution

5.9.6 Inception network

The primary goal from the inception network was to allow a model to learn the best hyperparameter choices for each layer of a network. Instead of manually choosing a filter size, the model will use three different filter sizes: 1×1, 3×3, and 5×5; in addition, each layer could also learn max pooling. To pick the best hyperparameters, the model concatenates the results from convolving with the different filter sizes and max pooling into a single output as shown in Figure 5.9.9a.

When we go deeper into the network we often run into a large number of channels with relatively small dimension size. Now consider, for example, a layer whose input is of size $28 \times 28 \times 192$ with a same convolution that has a $5 \times 5 \times 192$ filter size and 32 filters. In this scenario, our output will be of size $28 \times 28 \times 32$ and for each output, we will need $5 \times 5 \times 192$ total multiplications, which equates to around 120 million in total. Even with modern computers, 120 million multiplications will take a substantial amount of time.

Instead, if we first pass the input of size $28 \times 28 \times 192$ into a $1 \times 1 \times 192$ convolution with 16 filters and then pass that result into a $5 \times 5 \times 16$ same convolution with 32 filters we could end up with a pretty similar result with a far less computational cost. In total, we would have around 12 million total multiplications using the dimension reduction method we just proposed, which is shown in Figure 5.9.9b. Also, notice in our dimension reduction figure that we pass max pooling into a 1×1 convolution. This is because when using a max pooling filter, we cannot shrink the number of channels. Rather, we will shrink the number of channels with a 1×1 convolution, where the number of filters corresponds to the outputted channel size.



(a) Naïve version

(b) Dimension reduction

Figure 5.9.9: Inception layer (Szegedy et al., 2015)

The entire original inception network is shown in Figure 5.9.10 and was named GoogLeNet after the popular LeNet architecture.

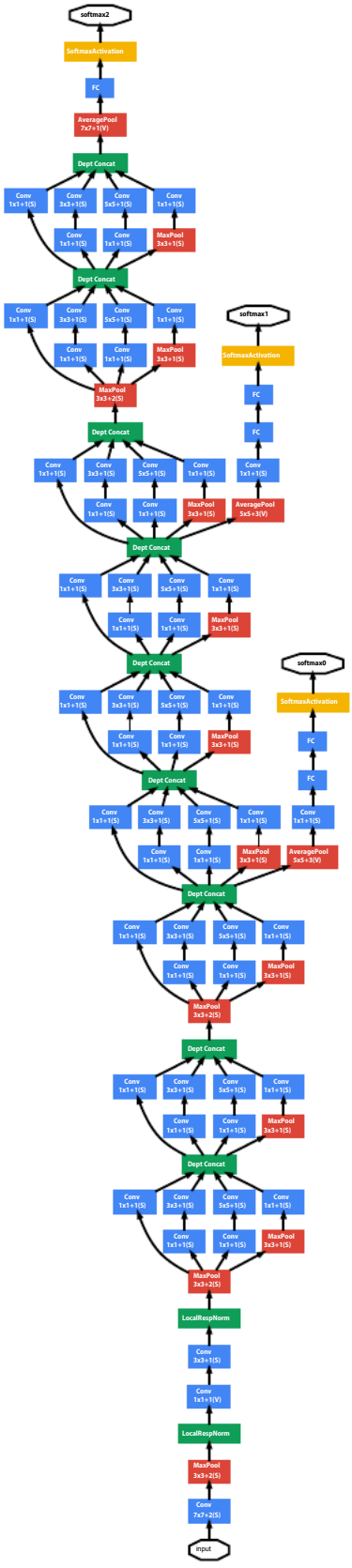


Figure 5.9.10: GoogLeNet architecture (Szegedy et al., 2015)

Figure 5.9.11: Meme cited in the original inception paper as a reasoning for the name inception network.

As a fun fact, the inception network cited the reasoning for its name came from the popular meme in Figure 5.9.11 that appeared in the movie Inception.

5.10 Competitions and benchmarks

To beat state-of-the-art models on a specific benchmark in competitions, there are a few methods employed that are rarely used in practice including ensembling and multi-crop at test time. Ensembling is when we test each instance of the testing set against a range of models, usually around 3-15, and use the prediction that either occurs most often (classification) or the average prediction (regression). Ensembling often performs better, but there is a significant computational cost that is introduced. The other method employed at test time is multi-crop which averages the output from different crops of the image. One common way to multi-crop an image take is known as 10-crop and is shown in Figure 5.10.1.

Figure 5.10.1: 10-crop includes mirroring an image and taking different croppings of both the image and its mirroring.

Chapter 6

Detection algorithms

6.1 Object localization

(a) Image classification **(b)** Classification with localization **(c)** Detection

Figure 6.1.1: Classification vs. localization vs. detection

Previously, we have only been working on image classification, which assumes only one object in a screen is present and determines that object. Next, we will be going into classification with localization which determines where the object is located in the image as shown in Figure 6.1.1b. We will also be going over image detection which allows us to classify multiple occurrences of the same *or* different classes from a single image as shown in Figure 6.1.1c.

With classification with localization, we are going to assume there is only one object in every image. As an example, we are going to walkthrough how a self-driving car could localize a **pedestrian** (1), **car** (2), **motorcycle** (3), or, if no object exists, a **background** (4). We can then change the output of the neural network from a softmax of 4 categories to include a bounding box with the center point, height, and width. The center of the bounding box will be denoted as (b_x, b_y) and the height and width will be b_h and b_w, respectively. Currently, we will be using square images and will label the top left corner as $(0, 0)$ and the bottom right corner as $(1, 1)$. The entire labeling of our image is shown in Figure 6.1.2.

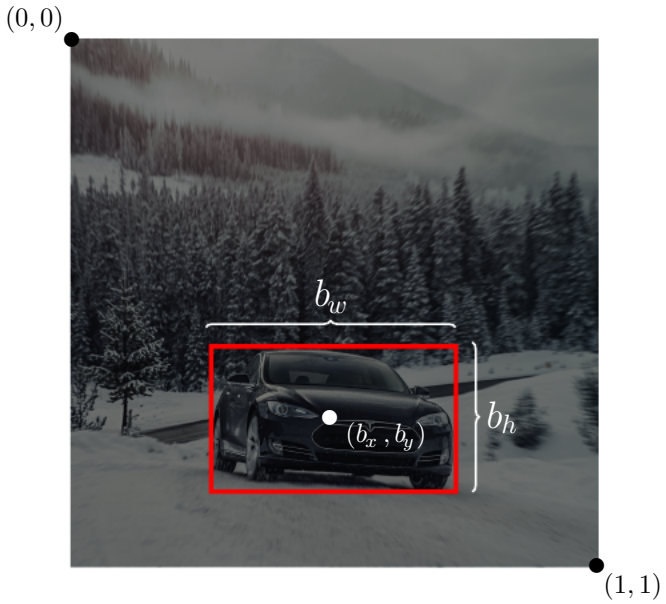


Figure 6.1.2: Our labeling for localization

We will also need to adjust our supervised learning labels to match the output of our model. We will define the output as

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ car? \\ pedestrian? \\ motorcycle? \end{bmatrix}, \tag{6.1.1}$$

where P_c is the probability we have a class 1, 2, or 3. A few labeled examples are shown in Figure 6.1.3.

$$y = \begin{bmatrix} 1 \\ 0.5 \\ 0.7 \\ 0.2 \\ 0.5 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$y = \begin{bmatrix} 0 \\ \text{None} \\ \text{None} \\ \text{None} \\ \text{None} \\ \text{None} \\ \text{None} \\ \text{None} \end{bmatrix}$$

(a) Car label **(b)** Background label

Figure 6.1.3: Our labels for different images

We will also define our loss function to be a squared error loss function

$$\mathcal{L}(\hat{y}, y) = \begin{cases} \sum_{i=1}^{8} (\hat{y}_i - y_i)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases}, \tag{6.1.2}$$

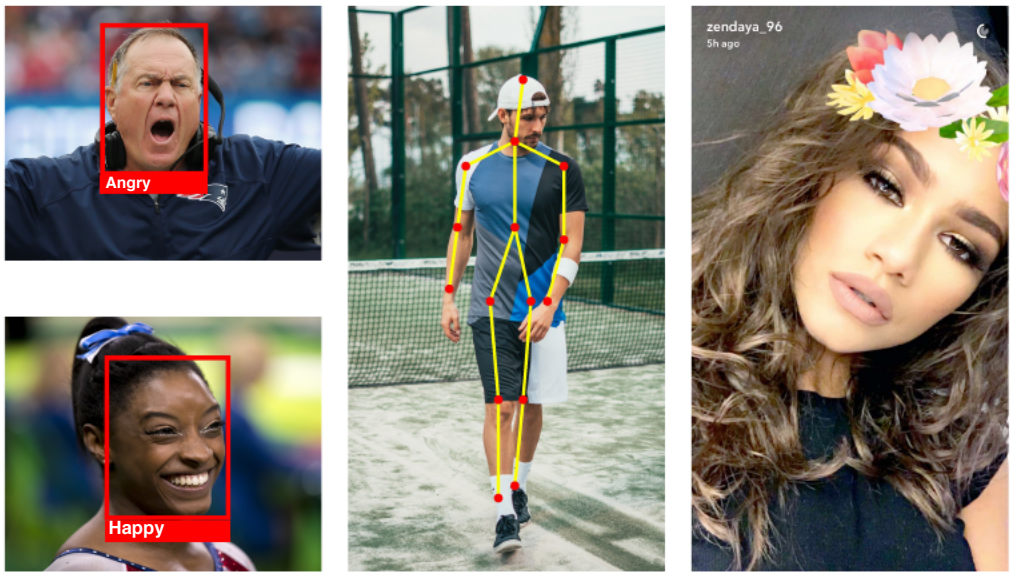
where the 8 comes from the size of our label.

6.2 Landmark detection

Landmark detection works to find specific areas of an image. Recently, landmark detection has been used to determine human emotions, determine the orientation of a moving body, and apply Snapchat filters. In order to create Snapchat face filter, like the one shown in Figure 6.2.1c, we can train a neural network to determine specific landmark locations inside of an image. To determine landmark locations on a face, we first need to classify a face and then label each landmark's location on our training data as shown in Figure 6.2.2, where each label with n landmarks would be in the form

$$y = \begin{bmatrix} face? \\ \ell_{1x} \\ \ell_{1y} \\ \vdots \\ \ell_{nx} \\ \ell_{ny} \end{bmatrix}, \tag{6.2.1}$$

where (ℓ_{ix}, ℓ_{iy}) is the xy coordinate of the ith landmark. We also must makr sure that we always label the ith landmark the same thing in each image. For example, if the first landmark is the leftmost part of somebody's left eye in one image, then it also must be the leftmost part of a person's left eye in another image.



(a) Emotion classification **(b)** Body tracking **(c)** Snapchat face filters

Figure 6.2.1: Applications for landmark detection

6.3 Object detection

Object detection focuses on locating multiple objects in a given image. A naïve approach to object detection is with the **sliding windows detection algorithm**. We will start with

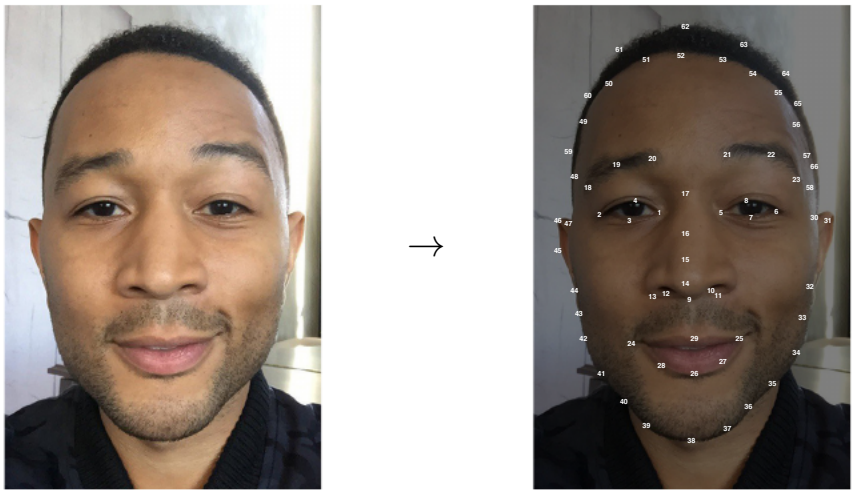


Figure 6.2.2: Adding a specific numbered landmark to each part of the image we are interested in

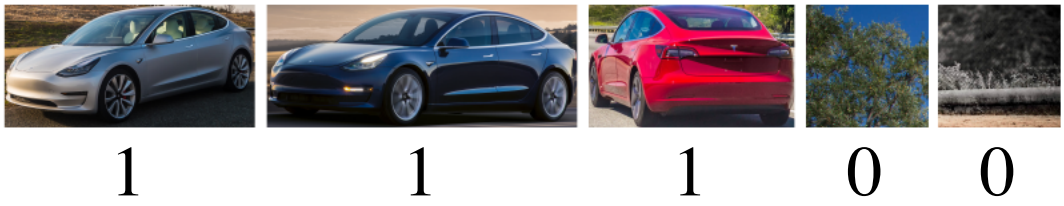


Figure 6.3.1: Sliding windows detection training set

a training set of tightly cropped images as shown in Figure 6.3.1. Then, we will create a window and slide it throughout our image. At each location where the window is on the image, only the pixels that the filter is covering will be passed into the convolutional neural network in hopes that the window will perfectly identify the location of objects. We will then change the size of the window and again repeat the sliding process as shown in Figure 6.3.2.

With the sliding windows detection algorithm just presented, we will run into a high computational cost in order to have an accurate image detector. Further, improving the algorithm by making the stride length shorter and testing more window sizes will increase the cost.

6.4 Sliding windows with convolution

To improve the computational cost of our sliding windows detection algorithm let us try to use a convolutional implementation where the output corresponds to sliding the window across each position. Consider taking a 3×3 matrix and convoluting it twice with a 2×2 filter, which will produce a 1×1 output as shown in Figure 6.4.1a. Now, consider if we were to add a row and column to the end of our input matrix, increasing it to a size of 4×4 and then convolute it twice with a 2×2 filter. What we end up with is a 2×2 output matrix where the upper left entry is equivalent to the two convolutions from the 3×3 input matrix as shown in Figure 6.4.1b. Similarily, the top right entry will be the two convolutions with the 3×3 input matrix without the first column and without the last row.

Using the method just introduced, we have a way to create a sliding window with a convolution. Taking this method a step further, we can also adjust the stride by adding a max pooling layer. The filter size of the max pooling layer will determine the size. Further, if we

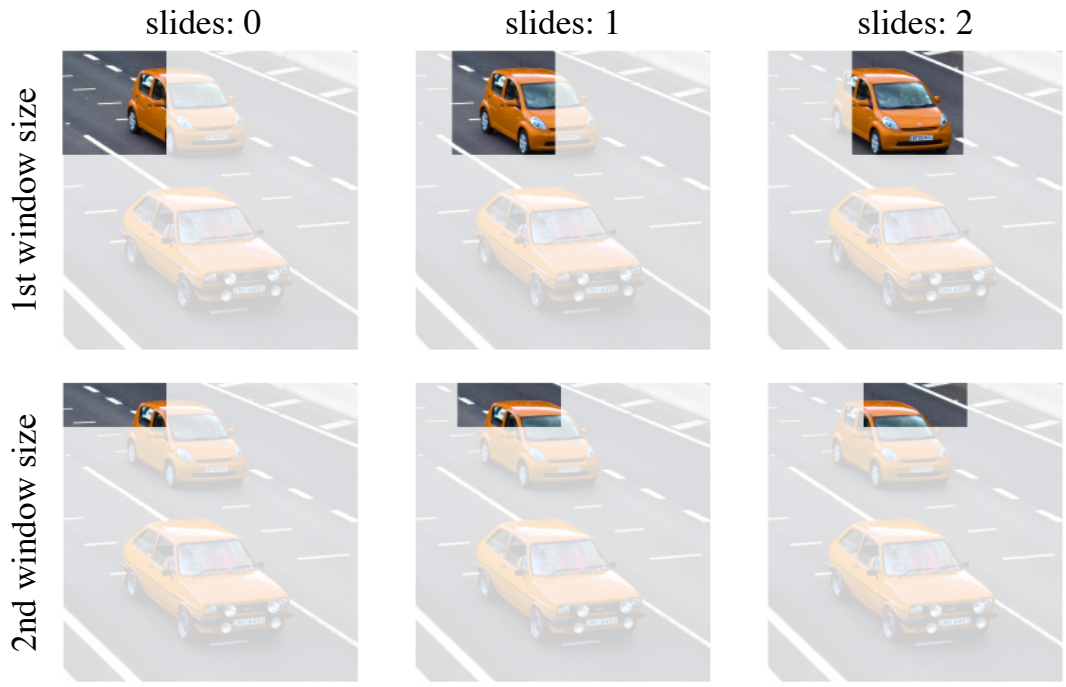
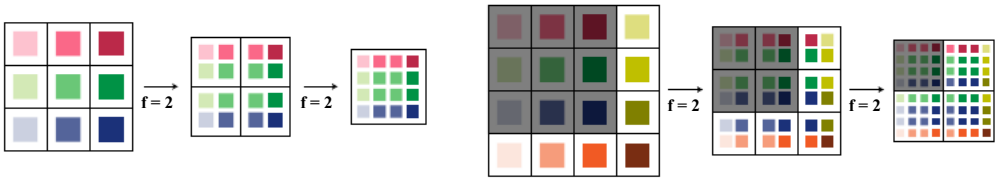


Figure 6.3.2: The sliding windows detection algorithm chooses different window sizes and slides the window size across the entire image. At each slide, the area that the window is covering is fed into a convolutional neural network in order to try and classify the image.



(a) 3×3 input **(b)** 4×4 input

Figure 6.4.1: Applying the same model to inputs of different size. The dark portion in Figure 6.4.1b is equivalent to the model in Figure 6.4.1a.

want to learn a more complex function we can add 1×1 convolutional neural networks to the end of our network that will act like fully connected layers as shown in Figure 6.4.2.

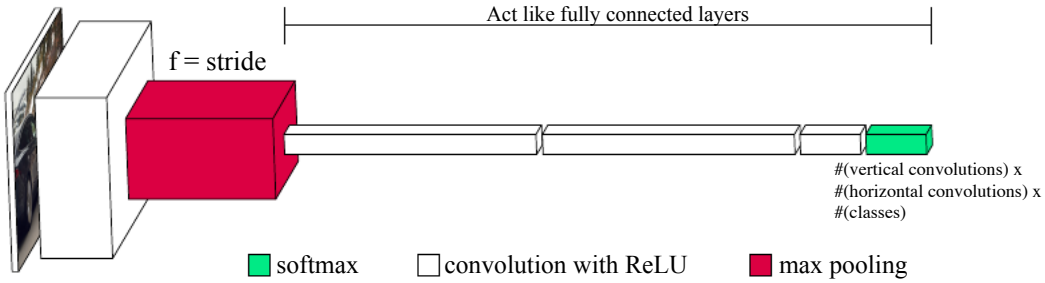


Figure 6.4.2: Convolutional neural network that determines if a window image is one of four categories: pedestrian, car, motorcycle, or background.

6.5 Bounding box predictions

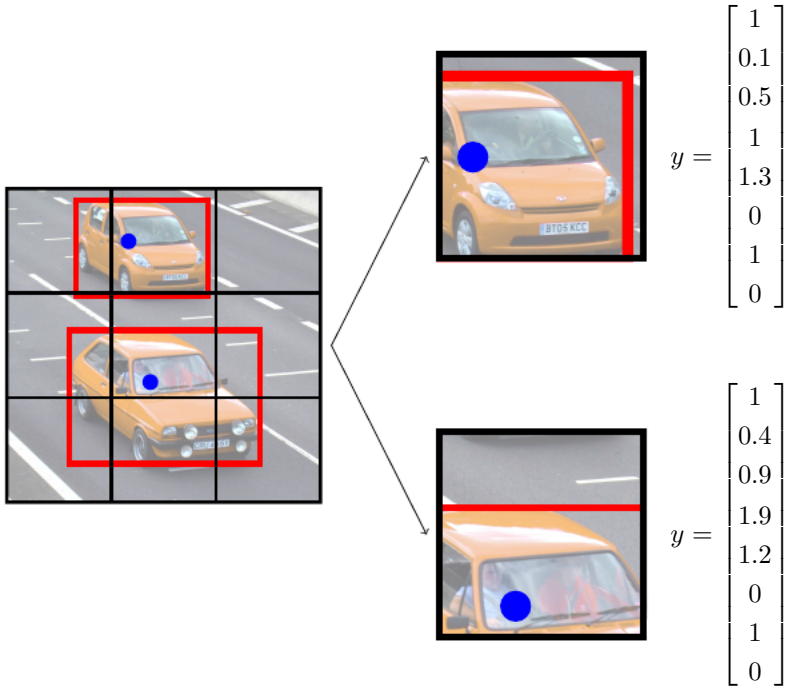


Figure 6.5.1: Using a bounding box on an image with the corresponding labels that contain cars. We are using the labeling set in Equation 6.1.1.

Currently, we have only seen guess and check algorithms in terms of choosing a filter and step size. Instead, it would be nice to only have one pass through a convolutional neural network that determines object detection. Let us consider breaking the image into a grid and treating each gridded cell as its own image that we have to label. If an object we are trying to identify has its centerpoint in the grid, then we will label P_c in that grid as 1. For grids that do not contain any objects or contain non-centerpoints of objects they will have P_c as 0 with `None` in all of the other values. When labeling images from the grids centerpoint, notice that our constraints are $0 \leq b_x, b_y \leq 1$ and $b_h, b_w \geq 0$. With our 3×3 grid, the output will need to be $3 \times 3 \times 8$ because each grid cell has its own label with 8 entries. Therefore, we can construct a convolutional neural network, like the one shown in Figure 6.4.2 that takes our image as input and outputs a $3 \times 3 \times 8$ tensor. The way we have set up bounding box prediction is known as the YOLO–You Only Look Once–algorithm and we will continue discussing it after introducing a few more ideas that play a key role.

6.6 Intersection over Union (IoU)

One common way to assess if a bounding box's prediction is accurate is to take the intersection of the labeled bounding box and the predicted bounding box and divide that by the union of the labeled and predicted bounding boxes, known as IoU. Typically if IoU ≥ 0.5, we say that our model's bounding box is correct, however, 0.5 is just an arbitrary choice and is sometimes changed.

6.7 Non-max suppression

When using smaller grid sizes for bounding box prediction, it is more likely that neighboring boxes will predict references to the same object as shown in Figure 6.7.1a. To start, we will first want to remove any boxes that are uncertain of containing a class. Typically, we may

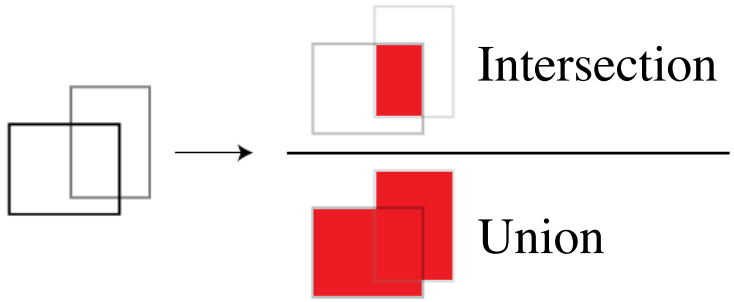
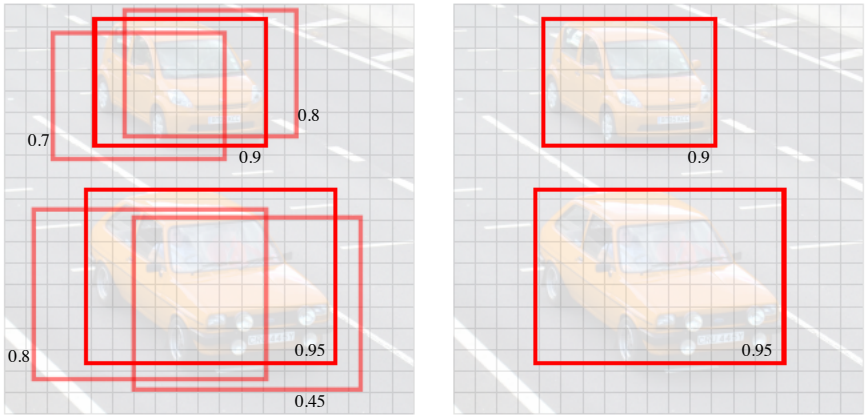


Figure 6.6.1: Taking the intersection over union of two bounding boxes to assess its correctness.



(a) Pre non-max suppression **(b)** Post non-max suppression

Figure 6.7.1: Predicted bounding boxes using a 19×19 grid along with the confidence a non-background class exists inside of that box.

remove all boxes that have $P_c \leq 0.6$. With a way to calculate the similarity between two bounding boxes (IoU), we can use that to determine if two bounding boxes are referring to the same object. Starting with the box that has the highest propability a class exists inside of it, P_c, we will keep that box and then remove any other box from the same class prediction that has an IoU ≥ 0.5 with the box in focus. Thus, we are effectively removing the non-maximum boxes from the prediction. The complete algorithm for non-max suppression is shown in Algorithm 8.

Algorithm 8 Non-max suppression

Input: `boxes` $\leftarrow$ list of all bounding boxes
Initialize: `finalBoxes` $\leftarrow$ empty list
Remove any box from `boxes` with $P_c \leq 0.6$
while `boxes` length > 0 **do**
  `maxBox` $\leftarrow$ box with max P_c from `boxes`
  Append `maxBox` to `finalBoxes`
  Remove `maxBox` from `boxes`
  for each `box` **in** `boxes` **do**
    if IoU(`box`, `maxBox`) ≥ 0.5 **then**
      Remove `box` from `boxes`
    end if
  end for
end while

6.8 Anchor boxes

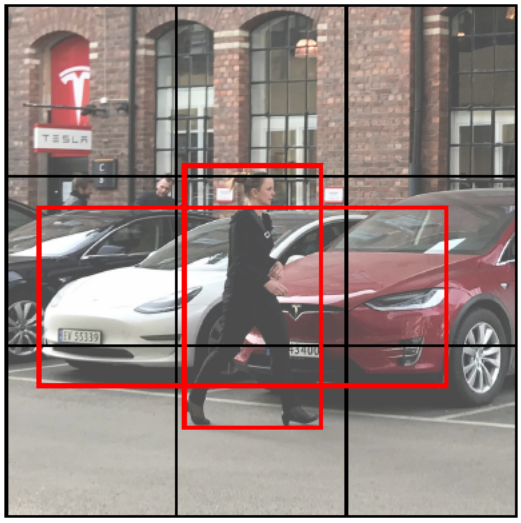
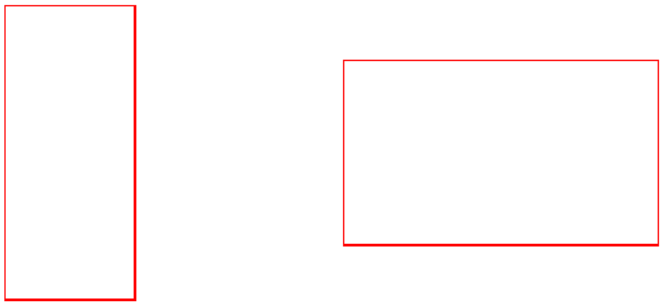


Figure 6.8.1: The midpoint for two objects–pedestrian and car–are located in the same grid cell.

With our current implementation for labelling each grid cell, we have no way to label two separate classes that occur in the same grid. For example, Figure 6.8.1 shows that the centerpoint for a pedestrian and a car occur in the center gridpoint. We will start off by a set number of different anchor boxes as shown in Figure 6.8.2. The sizes of the anchor boxes can be determined from a k-means clustering algorithm that looks for grouping of anchor box sizes amongst each class.



(a) Anchor box 1 **(b)** Anchor box 2

Figure 6.8.2: Defining multiple distinct anchor boxes

Now, for each label in our training data we will label it with

$$y = \begin{bmatrix} P_c \\ b_x \\ b_h \\ b_w \\ c_w \\ c_2 \\ c_3 \\ P_c \\ b_x \\ b_h \\ b_w \\ c_w \\ c_2 \\ c_3 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix} a \\ a \\ a \\ a \\ a \\ a \\ a \end{matrix}}\right\} \text{anchor box 1} \\ \\ \left.\vphantom{\begin{matrix} a \\ a \\ a \\ a \\ a \\ a \\ a \end{matrix}}\right\} \text{anchor box 2} \end{matrix} . \tag{6.8.1}$$

Then, for each grid cell we will use the IoU to determine which anchor box is closest to each bounding box. In Figure 6.8.1, the pedestrian would be asssigned to anchor box 1 and the car would be assigned to anchor box 2. We then fill in the output as the labels from each bounding box. With our 3×3 grid, the output size will be $3 \times 3 \times 16$ or $3 \times 3 \times (2 \times 8)$, since there are two anchor boxes and each anchor box has 8 entries. When using two anchor boxes, our prediction will give us 2 boundary boxes in each grid cell.

We could always define more anchor boxes in practice, alghough with a larger grid size, it is less likely for collisions in anchor boxes to occur. If we have more objects that share the same grid cell than we have anchor boxes or we have two boundary boxes that are the same dimension, our algorithm will not perform well.

6.9 YOLO algorithm

The YOLO (You Only Look Once) algorithm was introduced in 2016 by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. The algorithm combines boundary box predictions with anchor boxes and non-max suppression. With only having to move through the neural network in one pass, YOLO beat the running time for all previous state-of-the-art object detection algorithms.

Chapter 7

Sequence models

Sequence models allow us to work with variable inputs or variable outputs. This feature allows us to greatly expand the number of problems we are able to tackle. For example, sequence models are used for *speech recognition*, where given an variable length audio clip, we translate the spoken words; *music generation*, where we can input a set of data and output a sequence of music; *sentiment classification*, where we could take in a sentence review and output a star-rating; *DNA sequence analysis*, where we take a DNA sequence and label the part of the sequence corresponding to, say a protein; *machine (or language) translation*, where we are given a sentence in one language and have to convert it into another; *video activity recognition*, where we are given a set of frames and want to output the activity; and *name entity recognition*, where we are given a set of text and want to find the names within the text.

As an example, suppose we would like to build a sequence model for name entity recognition with the input

$$x : \text{Harry Potter and Hermione Granger invented a new spell.} \tag{7.0.1}$$

We will denote the tth word of the input as $x^{<t>}$, starting at index 1. For the output y, we will use multi-hot encoding where each word is labeled with a 1 or 0 as follows

$$y : 1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0. \tag{7.0.2}$$

The tth element of the output will similarly be referred to as $y^{<t>}$. The length of the input sequence will be denoted by T_x and the length of the output sequence will be T_y. In this case, both T_x and T_y are equal to 9, although that is not always the case. So, if we want to refer to the tth word in the ith element of the training sample, we will call $x^{(i)<t>}$. Since the input and output of the sequence can vary, we will use $T_x^{(i)}$ and $T_y^{(i)}$ to denote the length of the ith input and output, respectively.

To represent words, we will need use vocabulary list of all the possible words that are likely to occur. For modern applications, this could be around 50,000 words in total. We will represent teh vocabulary list sorted in ascending order. Then, to store $x^{<t>}$, we can use one-hot encoding, where we have a 1 in the index of the vocabulary list that stores the word we are looking at, and a 0 everywhere else. We will talk about *unknown* words, or words that are not in our vocabulary list, soon.

7.1 Recurrent neural networks

The problem with using a standard neural network is that the input length and output lengths can differ in different new samples of our data. If we tried to set a maximum number as the input, with 0 for each element that is not necessary, then we will not only greatly diminish the number of problems that we can focus on, but we also will be using a bunch of unnecessary memory in the network. The plain neural network will also not

take advantage of shared features of sequential data, similar to how convolutional neural networks took advantage of using the same filters over different regions of the image.

With recurrent neural networks, we built a model that takes as input the current time step t and all of the previous time steps in order to compute the output $y^{<t>}$. At each time step, the model is parameterized by shared weights W_{ax}, W_{aa} and W_{ya}, as shown in Figure 7.1.1. A big problem with this type of representation is that we are only looking at the previous time steps. Suppose we have the following sentences as input

$$\begin{cases} \text{He said, “Teddy Roosevelt was a great President.”} \\ \text{He said, “Teddy bears are on sale!”} \end{cases} . \tag{7.1.1}$$

In both of these cases, the first three words are equivalent; however, in only the first case does the word “Teddy” represent a name. We will delve into this topic soon, which looks at Bidirectional RNNs (BRNNs), however to understand the basics of RNNs we will stick with our unidirectional RNN.

For now, we will set $a^{<0>} = 0$, which is typical in many cases. Now, to represent the first activations and output in the recurrent neural network pictured in Figure 7.1.1, we will set

$$a^{<1>} = g\left(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a\right) \tag{7.1.2}$$

$$\hat{y}^{<1>} = g\left(W_{ya}a^{<1>} + b_y\right). \tag{7.1.3}$$

The activation functions in Equation 7.1.2 and Equation 7.1.3 do not have to be the same. With the activation functions, tanh and ReLU are commonly used, while sigmoid or softmax are typically used for the output activation functions. To generalize the previous equations, we have

$$a^{<t>} = g\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) \tag{7.1.4}$$

$$\hat{y}^{<t>} = g\left(W_{ya}a^{<t>} + b_y\right). \tag{7.1.5}$$

We will futher simplify Equation 7.1.4 to be

$$a^{<t>} = g\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right), \tag{7.1.6}$$

where W_a is formed by stacking

$$W_a = \begin{bmatrix} W_{aa} & | & W_{ax} \end{bmatrix} \tag{7.1.7}$$

into a single matrix. We will also use the notation

$$\left[a^{<t-1>}, x^{<t>}\right] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} \tag{7.1.8}$$

We will also simplify Equation 7.1.5 to be

$$\hat{y}^{<t>} = g\left(W_y a^{<t>} + b_y\right). \tag{7.1.9}$$

7.1.1 Backpropagation through time

We will define our loss at time step t as our familiar cross-entropy, or logistic loss

$$\mathcal{L}^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right) = -y^{<t>}\log\hat{y}^{<t>} - \left(1 - y^{<t>}\right)\log\left(1 - \hat{y}^{<t>}\right). \tag{7.1.10}$$

For the entire sequence, we will compute the loss as

$$\mathcal{L}\left(\hat{y}, y\right) = \sum_{t=1}^{T_y}\mathcal{L}^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right). \tag{7.1.11}$$

Now, we can define the computation graph pictured in Figure 7.1.2. When backpropagating through this network, we say that we are *backpropagating through time*, because we are going backwards in the time series.

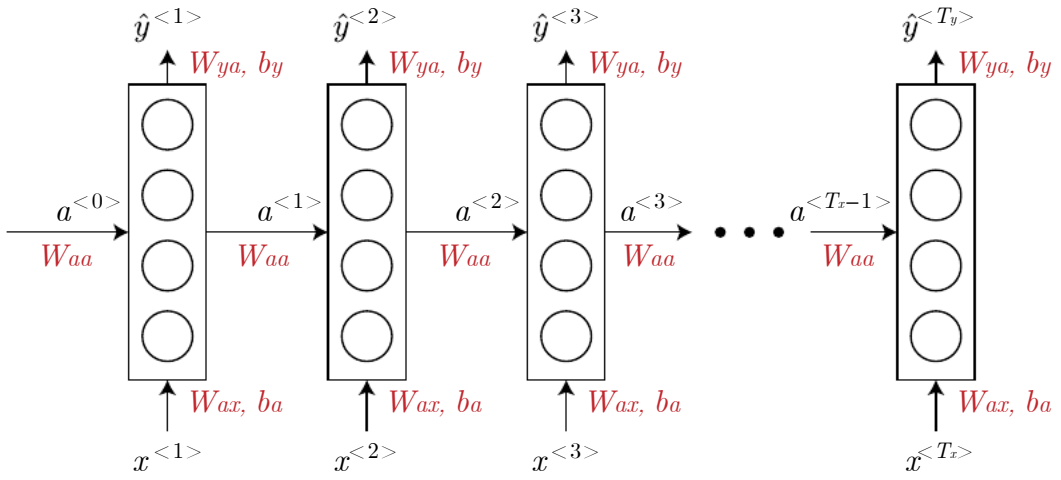


Figure 7.1.1: The structure of a basic recurrent neural network predicts $y^{<t>}$ based on all of the previous time steps, along with the current time step. The parameters of the network are pictured in red.

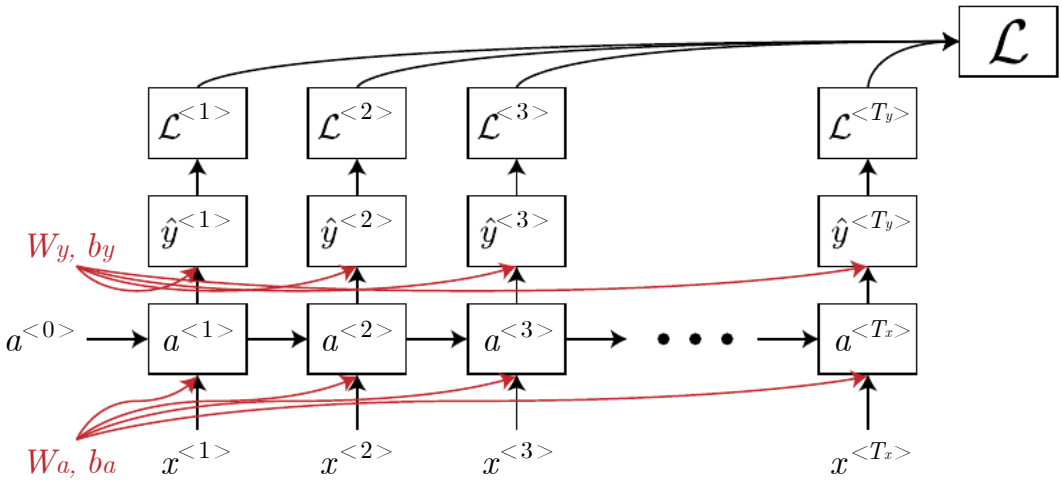


Figure 7.1.2: Computation graph for our recurrent neural network.

7.1.2 Variational RNNs

RNNs can be broken into different categories, as shown in Figure 7.1.3. What we have currently been working with is a many-to-many model. If we were to use sentiment classificaiton, where we input a review sentence and output the number of stars, we would use a many-to-one model. A one-to-many model may consist of music generation, where we could input some metadata (i.e. genre, style, etc.), and output scripts of music. Many-to-many models may also consist of examples where the number of inputs and the number of outputs differ, such as with machine translation.

7.1.3 Language modeling

Suppose we would like to build a speech recognition system that inputs an audio clip of speech and outputs the spoken words. Looking simply at the pronunciation of each word is not enough to determine the correct word that is spoken. For instance, the pronunciation for “pair” and “pear”; “two”, “to”, and “too”; and “their”, “they’re”, and “there” are all

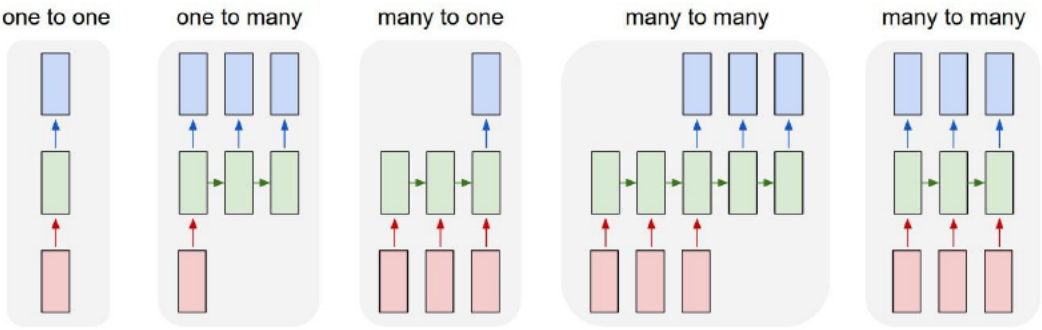


Figure 7.1.3: Depending on the combinations of fixed or variable length input and output, we can form different types of RNNs.

pronounced identitically, but are different words alltogether. A language model may look at a set of sentences that sound identitical and choose the sentence that is most likely to have occured. For example, suppose we have the two equivalently spoken sentences

$$\begin{cases} \text{The apple and pair salad.} \\ \text{The apple and pear salad.} \end{cases} . \tag{7.1.12}$$

Our language model may output something in the form of

$$\begin{cases} P(\text{The apple and pair salad.}) = 3.2 \times 10^{-13} \\ P(\text{The apple and pear salad.}) = 5.7 \times 10^{-10} \end{cases} , \tag{7.1.13}$$

thus, the second sentence would be selected because it is more probable.

To create a language model with a RNN, we need a training set that consists of a large *corpus* of English text. The word corpus comes from the field of natural language processing and refers to a set of text. For each sentence we are given as input, we will tokenize the output using one-hot encoding. It is also common to add an end-of-sentence token. If we are given a sentence like

$$\text{The Egyptian } \textit{Mau} \text{ is a bread of cat.} \tag{7.1.14}$$

and do not have the word "Mau" in our vocabulary list, then we will use a 1 to encode the unknown word item in our vocabulary list, while making everything else a 0.

Suppose we have a vocabulary list that has 10k entries (excludes puncuation and includes spots for unknown words) are working with the following sentence

$$x : \text{Well, hello there!} \tag{7.1.15}$$

With an RNN model that is trying to predict the next word, the inputs will be the ordered list of words that came before the current time step t. For the first time step, the input will be the 0 vector and the first prediction $\hat{y}^{<1>}$ will output a 10k softmax classification, where each word in the vocabulary list is given a probability that it occurs; that is

$$\hat{y}^{<1>} = P(< \text{each word} >) = \begin{bmatrix} P(< \text{word one} >) \\ \vdots \\ P(< \text{word 10k} >) \end{bmatrix} . \tag{7.1.16}$$

In this case, we have $y^{<1>}$ as a one-hot encoding with the word "well". For the second time step, our input will be $y^{<1>}$ = "well". Then prediction for $\hat{y}^{<2>}$ will be a conditional probability of the second word, given the first word; that is

$$\hat{y}^{<2>} = P\left(< \text{each word} > \mid y^{<1>} = \text{well}\right) . \tag{7.1.17}$$

For the third time step, the input will be $y^{<1>}$ = "well", $y^{<2>}$ = "hello". Following the same pattern, the prediction will be in the form of a 10k softmax classifier, where

$$\hat{y}^{<3>} = P\left(< \text{each word} > \mid y^{<1>} = \text{"well"}, y^{<2>} = \text{"hello"}\right) . \tag{7.1.18}$$

On the backward pass, we will look to maximize the predictions that

$$\begin{cases} \hat{y}^{<1>} = \text{well} \\ \hat{y}^{<2>} = \text{hello} \\ \hat{y}^{<3>} = \text{there} \end{cases} . \tag{7.1.19}$$

To predict the probability of an entire sentence, we could calculate

$$P\left(y^{<1>}\right) \cdot P\left(y^{<2>} \mid y^{<1>}\right) \cdot P\left(y^{<3>} \mid \bigcap_{t=1}^{2} y^{<t>}\right) \cdots P\left(y^{<T>} \mid \bigcap_{t=1}^{T-1} y^{<t>}\right). \tag{7.1.20}$$

7.1.4 Sampling novel sequences

To sample a sequence, we will use a similar RNN model to the one just described. However, we will now change the input at each time step to be all of the previous predictions from the model, as shown in Figure 7.1.4. We will also have the predictions be random variable choices from the probability distribution in order to avoid an infinite loop of the same few words. In NumPy, we can use `np.random.choice` to choose a random variable from the output prediction distribution. To end the program, we can wait until the sentence is over, in which case the output will be our `<EOS>` word. We could also iterate for a certain number of time steps or run a program for a specific amount of time.

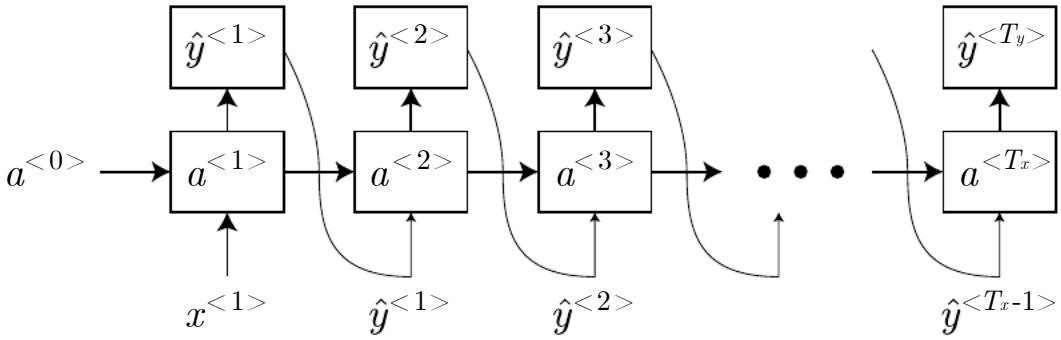


Figure 7.1.4: RNN model for sequence generation. Here, the to the time step t is the output from all of the previous time steps.

Up until this point, we have been focused on using a word vocabulary list; however, it is also common to use a character vocabulary list, such as all of the ASCII characters. An advantage of character level models is that we will never have to use an unknown token. One of the drawbacks with the character level approach is that it is worse at capturing long-range dependencies, such as opening and closing quotes, along with being more computationally expensive.

7.1.5 Vanishing gradients with RNNs

When we are working with RNNs, it is not unrealistic to be working with hundreds or thousands of time steps in a given model. However, recall the problem with deep neural networks that had a lot of layers, where we continued to multiply by smaller and smaller numbers as we backpropagate further. We can think of our the number of layers we have with NNs similar to the number of time steps with RNNs. With our current RNN approach, it will much harder to represent long-term dependencies in our data, such as opening and closing quotes.

7.1.6 GRU Gated recurrent unit

Currently, we have been using RNNs that have a hidden unit in the following form

$$a^{<t>} = \tanh\left(W_a \left[a^{<t-1>}, x^{<t>}\right] + b_a\right). \tag{7.1.21}$$

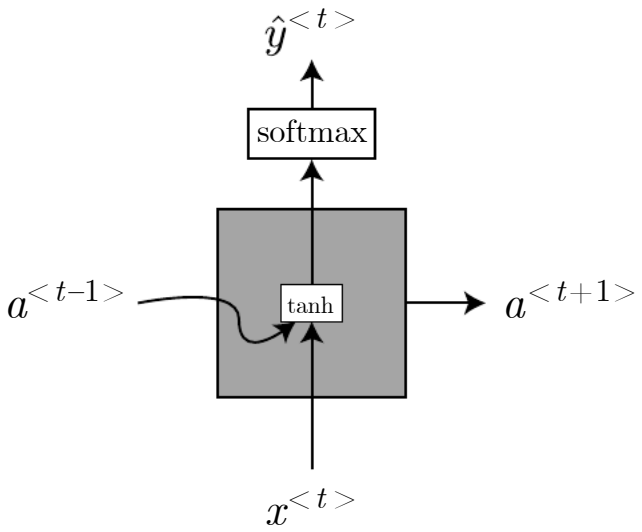


Figure 7.1.5: Our current approach to RNN calculations, where the gray box represents a hidden *black box* calculation.

We can picture this as a black box calculation at each time step t as shown in in Figure 7.1.5. However, our model is not adapt to capturing long-term dependencies in the data. Suppose we are working with the following sentence

$$x: \text{The } \textit{cat}, \text{ which already ate, } \textit{was} \text{ full.} \tag{7.1.22}$$

If we slightly changed this sentence to have the word cat as cats, then we would also need to update *was* to *were*. With a GRU, $a^{<t>}$ will be treated like a *memory cell*. We will then set

$$\tilde{a}^{<t>} = \tanh\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right) \tag{7.1.23}$$

as a potential candidate for $a^{<t>}$. Next, we will create an *update gate*

$$\Gamma_u = \sigma\left(W_u\left[a^{<t-1>}, x^{<t>}\right] + b_u\right), \tag{7.1.24}$$

where the majority of values in Γ_u will be 0 or 1, based on the functionality of the sigmoid function. The update gate ultimately determines if we will update $a^{<t>}$. For example, suppose the model learns that singular words are denoted with $a^{<t>} = 1$. Then, in our working sentence we may have something along the lines of:

t	1	2	3	4	5	6	7
x	The	cat	which	already	ate	was	full
$a^{<t>}$	–	1	1	1	1	1	–

Notice that the update gate continues to denote that our sentence is singular at further time steps t. At some future time t, if we run into a plural phrase, then it is the job of Γ_u to update $a^{<t>}$. Specifically, our update will be

$$a^{<t>} = \Gamma_u * \tilde{a}^{<t>} + (1 - \Gamma_u) * a^{<t-1>}, \tag{7.1.25}$$

where $*$ is an element-wise multiplication between the vector. Notice here that when $\Gamma_u = 1$, we will update $a^{<t>}$ to be $\tilde{a}^{<t>}$, such as when the time step was 2 in the above example. When $\Gamma_u = 0$, then we keep $a^{<t>}$ the same value, such as in the time steps [3,6] above. An important note is that the values $\tilde{a}^{<t>}$, $a^{<t>}$, and Γ_u are all vectors that can learn many different types of long-term dependencies. For example, we can learn how to open and close brackets, quotes, and use singular or plural phrases when appropriate. The computation graph for GRUs is shown in Figure 7.1.6.

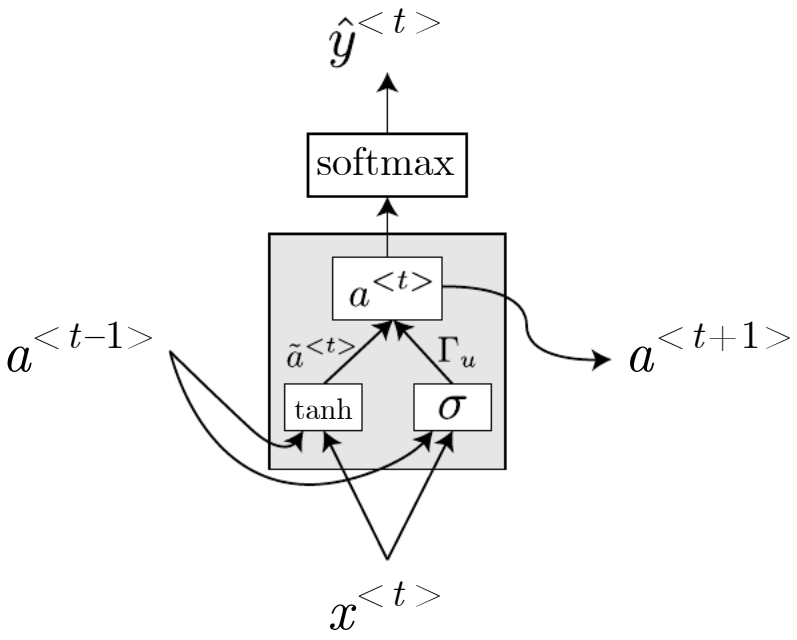


Figure 7.1.6: Simplified GRU computational graph at a time step t.

There is one other part to the GRU that we have currently left out, and that is to determine how *relevant* the last memory cell $a^{<t-1>}$ is for the calculation of the next, potential, memory cell $\tilde{a}^{<t>}$. Here, we add the relevance gate in as follows

$$\tilde{a}^{<t>} = \tanh\left(W_a \left[\Gamma_r * a^{<t-1>}, x^{<t>}\right] + b_a\right) \tag{7.1.26}$$

$$\Gamma_u = \sigma\left(W_u \left[a^{<t-1>}, x^{<t>}\right] + b_u\right) \tag{7.1.27}$$

$$\Gamma_r = \sigma\left(W_r \left[a^{<t-1>}, x^{<t>}\right] + b_r\right) \tag{7.1.28}$$

$$a^{<t>} = \Gamma_u * \tilde{a}^{<t>} + (1 - \Gamma_u) * a^{<t-1>}. \tag{7.1.29}$$

7.1.7 LSTM: Long short-term memory

In addition to GRUs to store long-term dependencies in sequential data, LSTMs are also commonly used. Figure 7.1.7 shows an example of an LSTM model. We can think about LSTMs as an extension of GRUs with 2 hidden units $\mathbf{h}^1$ and $\mathbf{h}^2$. We use 3 separate gates, a forget gate Γ_f, update gate Γ_u, and output gate Γ_o, that are each defined as

$$\Gamma_u = \sigma\Big(\mathbf{W}_u \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\Big) \tag{7.1.30}$$

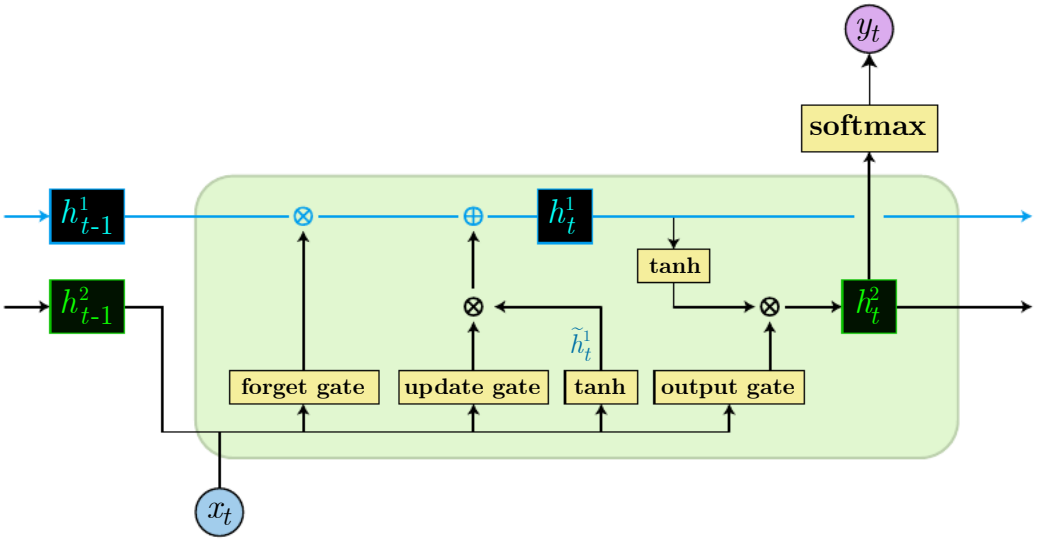
$$\Gamma_f = \sigma\Big(\mathbf{W}_f \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\Big) \tag{7.1.31}$$

$$\Gamma_o = \sigma\Big(\mathbf{W}_o \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]\Big). \tag{7.1.32}$$

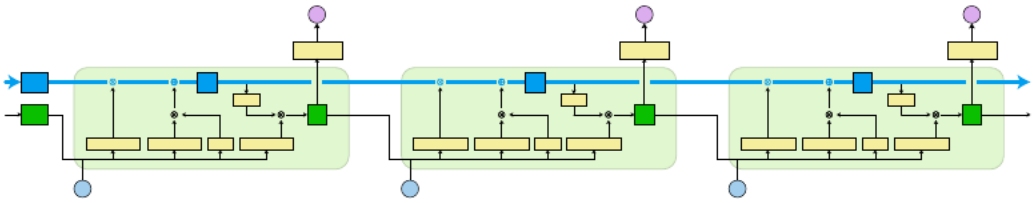
With LSTMs, we remove the relevance gate Γ_r because it often does not improve the performance of a model in practice. With our 1st hidden unit $\mathbf{h}^1$, we will set its replace candidate based on the other hidden unit input $\mathbf{h}^2_{t-1}$ and the new input $\mathbf{x}_t$, which we will define as

$$\tilde{\mathbf{h}}^1_t = \sigma\Big(\mathbf{W}_c \cdot [\mathbf{x}_t, \mathbf{h}^2_{t-1}]\Big). \tag{7.1.33}$$

Now instead of using a single update gate that determines both how much of the previous layer we want to forget and how much of the replacement candidate we want to remember,



(a) Single LSTM gate



(b) Stacking multiple LSTM gates in a row

Figure 7.1.7: LSTM model

we use 2 separate gates to define these quantities. In particular, we define our 1st hidden unit to be

$$\mathbf{h}^1_t = \Gamma_u \otimes \tilde{\mathbf{h}}^1_t + \Gamma_f \otimes \mathbf{h}^1_{t-1}. \tag{7.1.34}$$

Our 2nd hidden unit is dependent on $\mathbf{h}^1_t$ and passed into the softmax fucntion to produce the output y_t, which we define as

$$\mathbf{h}^2_t = \Gamma_o \otimes \tanh\left(\mathbf{h}^1_t\right). \tag{7.1.35}$$

Here, the output gate determines the relevance of each hidden unit for the output. The blue path that we show in Figure 7.1.7b connects the 1st hidden unit between multiple LSTM gates. The path could give an intuitive feel for why LSTMs give long-range dependency connections, which is primarily due to having multiple update gates control what we store in the internal memory, making it harder to forget everything completely.

7.1.8 BRNNs: Bidirectional RNNs

Recall our 2 sentences we used earlier

$$\begin{cases} \textit{He said, "Teddy} \text{ bears are on sale!"} \\ \textit{He said, "Teddy} \text{ Roosevelt was a great President!"} \end{cases}, \tag{7.1.36}$$

where it is not enough to predict if the work *Teddy* is a name based on the information before the word occurs. In this case, we can use a *bidirectional RNN* (BRNN) to make predictions with the words before and after *Teddy*. Figure 7.1.8 shows the diagram for a BRNN with a fixed size input length of 4. Our BRNN is broken up into 2 separate forward RNNs. Here, we have a forward RNN with activations in the form $\overrightarrow{a}$ that takes into account all of the previous time step inputs. We also have another RNN with activations in the form of $\overleftarrow{a}$

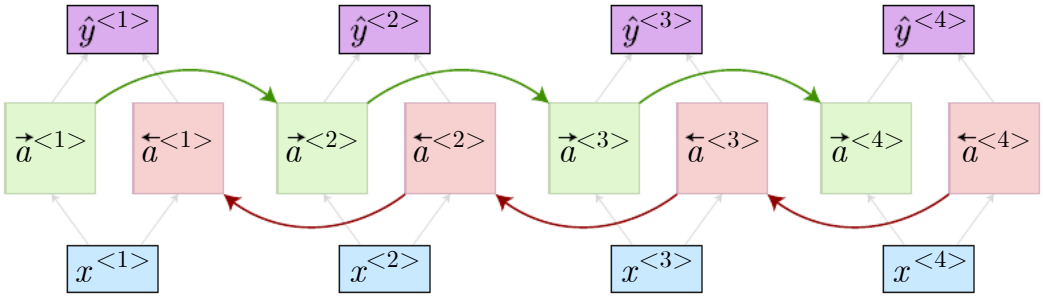
that looks at all the future time steps in reverse order. For example, if our sentence is

$$\textit{He said, "Teddy Roosevelt!"}, \tag{7.1.37}$$

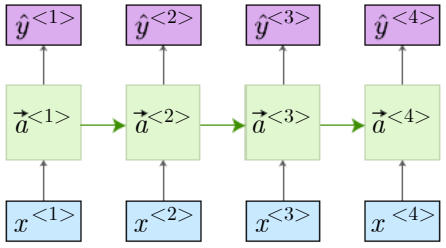
then at $t = 3$ we can form $\overleftarrow{a}^{<3>}$ with $\{\textit{He, said, Teddy}\}$ and we can form $\overrightarrow{a}^{<3>}$ with $\{\textit{Roosevelt}\}$. We can then make an output prediction, such as the meaning of the word *Teddy*, in the form of

$$\hat{y}^{<t>} = g\left(\mathbf{W} \cdot \left[\overrightarrow{a}^{<3>}, \overleftarrow{a}^{<3>}\right]\right). \tag{7.1.38}$$

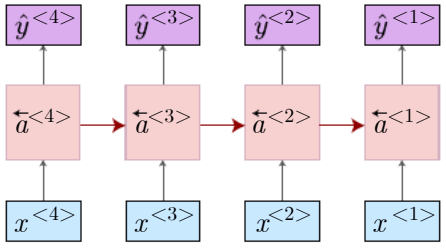
One of the downsides to using a BRNN is that we are required to supply the entire input at the start.



(a) Forward and backward connections



(b) Forward RNN connection with the sequence in order



(c) Forward RNN connection, where the sequence is in reverse order

Figure 7.1.8: BRNN with an input of length of 4

7.1.9 Deep RNNs

Figure 7.1.9 shows the 2 primary types of deep RNNs, where we stack multiple activations at each time step. The 1st type of network, in Figure 7.1.9a, shows 3 stacked activations at each time step that are each connected to the next time step. Here, each layer has its own weights, which are shared at each time step. If, for example, we can calculate each hidden activation in layer ℓ on the tth time step, then we can use

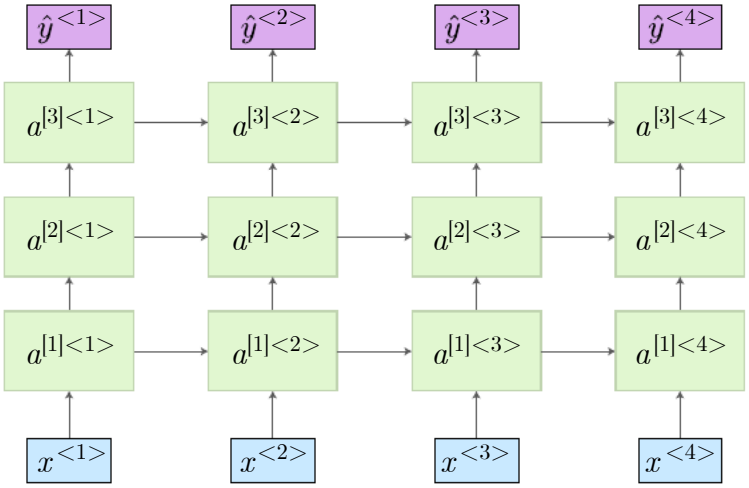
$$a^{[\ell]<t>} = g\left(\mathbf{W}^{[\ell]} \cdot \left[a^{[\ell]<t-1>}, a^{[\ell-1]<t>}\right]\right), \tag{7.1.39}$$

which takes the left and below activations as the input. The 2nd type of network, in Figure 7.1.9b, shows the first few layers connecting to future time steps, while the deeper layers are only connected to a single time step. Here, the deeper layers are not connected to future time steps because connecting them would be computationally expensive.

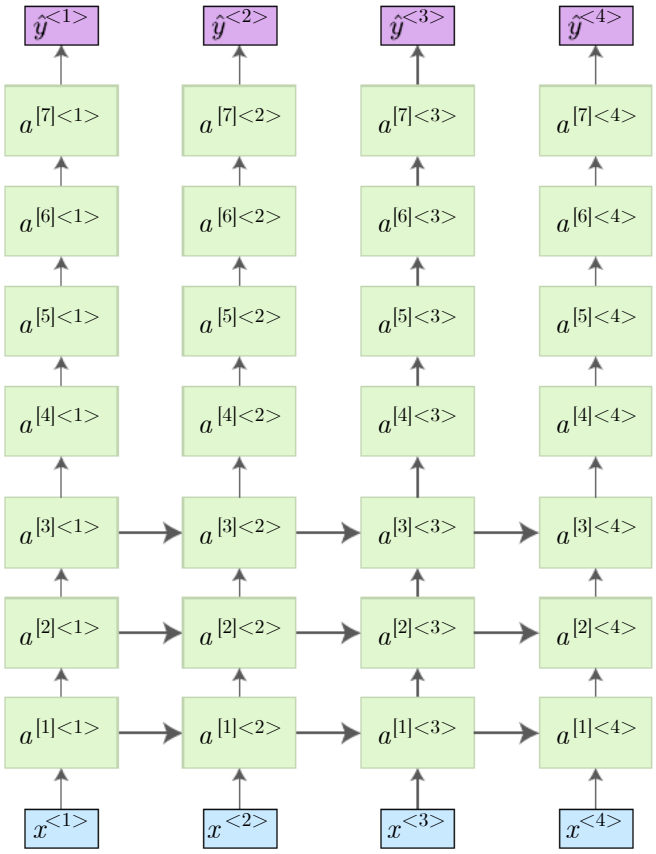
7.2 Word embedding

7.3 Word representation

Up until this point, we have represented words with a dictionary, where we specify a specific word using a one-hot encoding vector. One of the problems with this representation is that



(a) Stacking together multiple layers of activations that are each connected to the next and previous time steps



(b) Deep RNN where the first few layers have activations that connect to future time steps, while the deeper layers only carry weight in a single time step

Figure 7.1.9: Deep RNNs

the words do not have a connection with each other. For example, suppose we input

I want a glass of orange _____ (7.3.1)

input our network, where it is the goal of the network to predict the blank. With a well trained network the likely prediction is the word *juice*. But, suppose we now have the input

$$\text{I want a glass of apple } \_\_\_\_\_ \tag{7.3.2}$$

and our model does not know about apple juice. In this situation, if our model knows that there is some connection between orange and apple, then it would still likely predict juice.

One approach to learning connections would be to discover feature representations with each word. Table 7.1 shows an example of a potential encoding that our model may find. The table shows the words *king*, *queen*, *apple*, and *orange*, along with their responses to a set of features. For example, if we have a *royal* feature, then *king* and *queen* might have a high response; alternatively if we have a *food* feature, then *apple* and *orange* might have a high response. Here, if we have 300 features, then we can represent each word with a vector of length 300. Now, due to the similarities between features, our network is more likely to pick up similar words.

	King	**Queen**	**Apple**	**Orange**
Royal	0.93	0.95	-0.01	0.00
Age	0.7	0.71	0.03	-0.02
⋮	⋮	⋮	⋮	⋮
Food	0.01	-0.03	0.95	0.97

Table 7.1: Sample words in a dictionary (top) with their responses to each feature (left) that ranges between $[-1:1]$

7.3.1 Using word embeddings

To learn the features for a set of words, it is common to use transfer learning. There are currently many freely available text corpuses that have learned word embeddings training on 1B to 100B words. Once we have word embeddings, we can transfer the words to a smaller task and fine-tune the parameters if necessary.

7.3.2 Properties of word embeddings

Once we have the word embeddings, one neat thing that we can find is the difference between 2 words. Suppose for example that one of our features gives the value 1 for words describing objects with an orange color, -1 for words describing a colored object that is not orange, and 0 for objects that are not defined by a color. Table 7.2 shows how we can spot the feature differences between the *apple* and *orange* vectors. In particular, if we subtract the 2 vectors, then the features that are largest in magnitude correspond to the feature differences between words, while the feature differences near 0 refer to features being the same.

	Apple	**Orange**	**Apple - Orange** (approximate)
Orange colored	-0.96	0.98	-2
Royal	-0.01	0.00	0
Age	0.03	-0.02	0
Food	0.95	0.97	0

Table 7.2: To compare words we can find the differences and similarities by taking the difference between the word feature vectors, and finding the features with the largest difference in magnitude are the most different, while feature differences near 0 are quite similar.