Abstract

Dewayne A. Dixon

Neural Network and its Application to an Image Binary Classification Problem

(Under the direction of Dr. Yongjin Lu, Dr. Wei-Bang Chen and Dr. Tariq Qazi)

We train artificial neural networks to classify cat and non-cat images pulled from the CIFAR-10 dataset. The data set will be separated into a training set and a test set with equal distribution of the images into the true and false class. The binary classifier will be trained using different architectures of neural network, which include shallow neural network (neural network with one hidden layer) and convolutional neural networks. The performance of these algorithms is evaluated based on the prediction accuracy on the train set and test (cross validation) set. Mathematical details behind the main ideas of these algorithms will be discussed as well.

Acknowledgment

This is dedicated to my brother, Davyn A. Dixon, who inspired me throughout this entire process. I would like to thank Dr. Y. Lu for his patience God for his many blessings.

Table of Contents

Chapter 1

Introduction: Deep Learning

Deep learning systems have transformed many major industries, for example marketing, real estate, health insurance, etc. Artificial Intelligence (AI) is described as being the new electricity. As history has told, electricity has major impact on major industries. AI is currently having the same impact on industries, presently, and it is rapidly growing.

1.1 History

The origins of artificial neural networks were to create algorithms that try to mimic the brain. Neural networks were first modeled in a paper, simplistically utilizing electrical circuits, in 1943 by a mathematician and neurophysiologist, Walter Pitts and Warren McCulloch, respectively.(McCulloch et al.) *The Organization of Behavior*, written by Donald Hebb in 1949, points out that neural pathways are strengthened every time it's implemented.(Hebb) It became possible, in the 1950s, to simulate a hypothetical neural network, firstly by Nathaniel Rochester, which failed.(Wikipedia) In 1959, "ADALINE" and "MADALINE" were developed, by Bernard Widrow and Marcian Hoff, to recognize binary problems. MADALINE was used as the first neural network implements in a real world problem which is still in commercial use.(Widrow) In 1962, Widrow and Hoff developed a learning procedure that examines the value before the weight adjusts it. Despite later success, the traditional von Neumann architecture became more researched, leaving behind neural networks. A paper proposed the idea that there could be an extension of a single layered neural network to a multiple layered network. In 1975, the first multiple layered neural network was developed considered an unsupervised network. It was not until the late 1990s when neural networking regained research interest.

1.2 Basic Structure

To understand fully how a neural network operates, one could take note of the biological function of parts of a single neuron, in which neural networks duplicate.

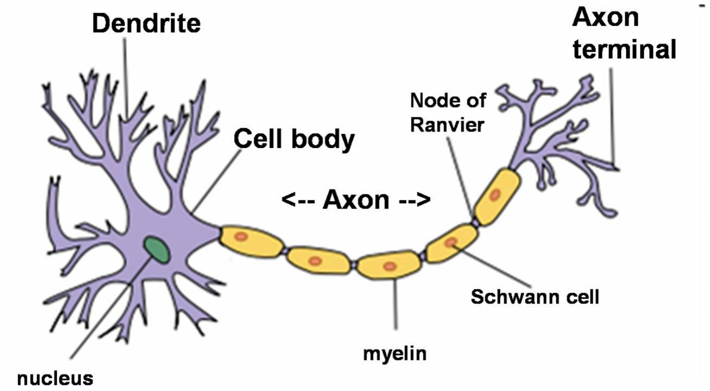


Figure 1.1: This figure shows the parts on a neuron.

Neural networks mimic the function of a brain's neuron. There are about 10^{11} neurons in the brain where majority of the neurons lie in the cortex of the brain. The neuron itself is a the fundamental structure and functional unit of the nervous system. There are four major components of the neuron: the dendrites, soma (or the cell body), axon and the synapse. The dendrite receives input information from other neurons through the synapse. The synapse is the extracellular compartment of the brain which contain the output signals from neighboring neurons. The signals are then processed in the soma, or the cell body. Once the cell body determines the response of the signal, it then transmits that response through the axon. The neuron then sends the output into the synapse where the response is enacted.

Figure 1.2 is an example of a basic neural network structure. For future use, we use the following notations: given a set of data, which we call a training set, containing m number of training examples, denoted by (x_i, y_i) where $i = 1, 2, ..., m$. The dendrite can be considered as the "input wires" of the neural network. Here, we take the input variables, denoted $x(i)$ and

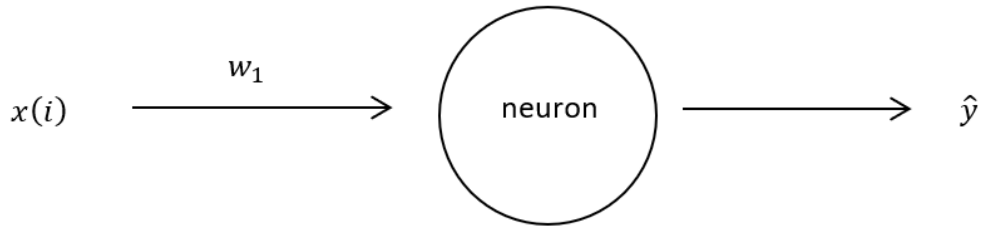


Figure 1.2: This figure shows the parts of a basic model of a neural network.

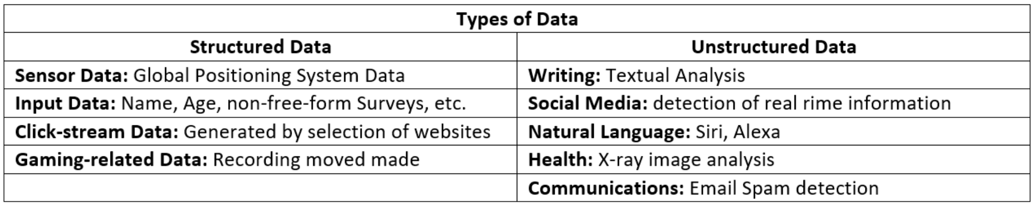
also called input features, and process them in first layer of the network. These features are then sent to the cell body of the neural network through weights, denoted w_i where $i = 1, 2, ..., m$, which is considered the hidden layers or learning algorithm, where data is processed. Weights adjusts as the learning algorithm proceeds, increasing or decreasing the strength of the signal at a connection. After the hidden layers process the signal, a final layer, the output layer then gives the response, called output variables to the algorithm giving us our hypothesis/error, $\hat{y}$. Using X to denote the training set and Y to denote the output values, we can learn a function $\hat{y} : X \to Y$ where $\hat{y}_i$ is considered to be a good prediction for the corresponding value of y_i.

This process mimics its biological counterpart: take into consideration that the neuron's cellular membrane has relative selectivity, which means that only certain ions can cross into the cellular membrane ionic channels. Signals are then directed from one neuron to the next through a complex chemical transmitting process called synapse propagation. That filtered information enters the dendrite where it's changed in the cell body. Once the cell membrane is above the threshold, the action potential is sent through the axon. Once the action potential is released in the synapse, the neuron release biochemicals that have an inhibitory effect (resting potential) or excitatory effect (action potential) on neighboring neurons.

1.3 Data Mining and Machine Learning

Artificial neural network belongs to a broader branch of data science called data mining or machine learning which uses algorithms that continuously learn from data. In data mining/machine learning, data can be represented as an $n \times d$ data matrix where n are the number or rows and d are the number of columns. We can arrange the matrix in two ways: the rows can be the example and columns be the feature, or the arrangement,which we will utilize, where the columns represent the example and the rows represent the feature. The attributes, or elements in the data matrix, can be classified as either numerical attributes with real-valued domain (e.g. Temperature, Age, Shoe Size) or categorical attributes with set-valued domain composed of a set of symbols (e.g. Sex, Race, Education Level).

The data could be roughly classified as structured data and unstructured data: **Structured data** refers to data with a high degree of organization. This includes defining what type of data will be stored as the input value and how it will be stored; for example, the data type may be numeric, currency, alphabetic, names, dates, temperatures, etcetera. We also have to include any restrictions or constraints on the data input; for example, number of characters and restricting certain terms and ranges. For example, a student's grade output with respect to minutes studied is structured. **Unstructured data** is all those things that can't be readily classified, meaning there are relatively no restrictions on the input data. This data doesn't reside in the usual row column database. For example, email messages with classification, word processing documents, videos, photos, etcetera.



Types of Data	
Structured Data	**Unstructured Data**
Sensor Data: Global Positioning System Data	**Writing:** Textual Analysis
Input Data: Name, Age, non-free-form Surveys, etc.	**Social Media:** detection of real rime information
Click-stream Data: Generated by selection of websites	**Natural Language:** Siri, Alexa
Gaming-related Data: Recording moved made	**Health:** X-ray image analysis
	Communications: Email Spam detection

Figure 1.3: Structured vs Unstructured Data Examples

Depending on how the data is processes, machine learning/data mining falls into the following categories: supervised, unsupervised, semi-supervised and reinforcement. In supervised learning, the algorithm is trained by inputting values with the corresponding correct output values. Once the training data set is complete on the algorithm, we check for potential error. If there is error, we have to minimize the error of the model through refinement, **gradient descent**. An example of supervised learning would be given a collection of images, we want to identify which of them are cats; or a collection of molecules that are current drugs where we are able to train an algorithm to detect is a new collection is also a drug.

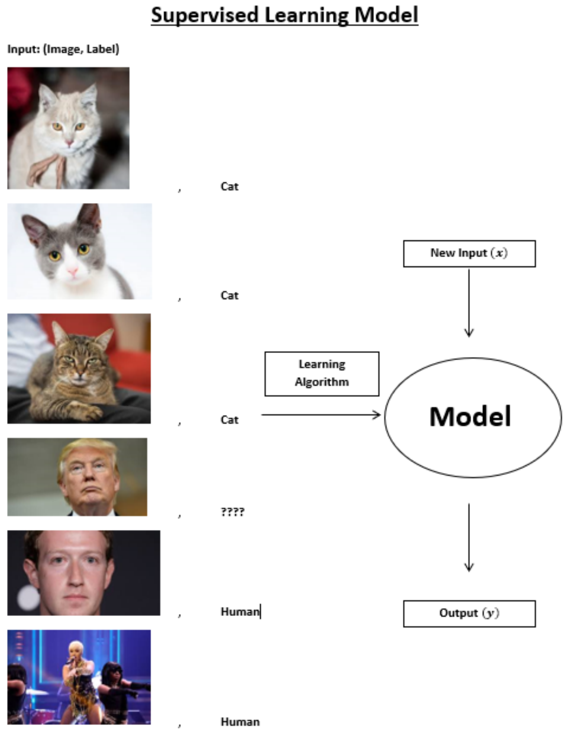


Figure 1.4: Example of Supervised Learning Model

There are two types of supervised learning: regression and classification. In regression problems, the prediction of results being within a continuous output. Simply, we are mapping your input features to a continuous output function. Simply, we are mapping your input variables to a continuous output function. In classification problems, we input variables into the algorithm obtaining discrete categories, resulting in a discrete output.

In unsupervised learning, the algorithm must explore the data and detect hidden patterns or structure within the data. There are two methods that are used in unsupervised learning:

- Principal components analysis: a tool for data visualization or data pre-processing before supervised techniques are applied.

- Clustering: a method for discovering unknown subgroups in data.

Given a bunch of photos of m people without knowing who is who we want to separate the photos into m piles with containing only one individual is an example of unsupervised learning.

Semi-supervised learning utilizes both labeled and unlabeled data to train the algorithm (Supervised and Unsupervised data sets combined) where reinforcement learning deals with trial and error to determine which action yields the highest reward, used frequently in gaming.

To train ANN, the label on the response variable should be given for each example. This is exactly a requirement of supervised learning. Thus, we will follow the standard procedure of supervised learning to train ANNs: data extraction, data cleaning, data fusion and then feature construction. We then are able to choose an algorithm that will best fit the model. Once we update the parameters to convergence, we then utilize the test set to validate our algorithm. We must use the post-processing sets where we interpret and the pattern and model. We then confirm the hypothesis which will allow us to generate the outputs.

1.4 Data and Methodology

We begin with two data sets; testing and training sets. We wanted to create an ANN which binary classifies images into to cat or not cat. The prediction obtained would then be tested for accuracy utilizing a testing set of 1800 examples where 50% were cat. We began with 7200 training examples where 50% were cat images of size $32 \times 32 \times 3$. We start the building of the algorithm by first exploring the accuracy of Linear Regression and Logistic regression, which we found did not produce accurate predictions or performed very slowly. From there, we explored the accuracy of shallow Artificial Neural Network (SANN) which produced a more accurate prediction. We then explored the accuracy of two Convolutional Neural Networks (CNN) that are utilized often with predictions, AlexNet and VGG-16. We found that both CNNs produced more accurate predictions than SANN, but also VGG-16 was slightly more accurate than AlexNet. Concluding that the binary classification prediction is more successful utilizing the VGG-16 CNN with a 83.28% accuracy where AlexNet and SANN accuracy were 81.83% and 66.61%, respectively.

The rest of the thesis is organized as follows: in Chapter 2, we discuss the definition of image and describe the images we will be processing; in Chapter 3, we define and observe the performance of a basic Artificial Neural Network with no hidden layers; in Chapter 4, we introduce the common activation function utilized in ANN and define and observe the performance of an ANN with hidden layers; in Chapter 5, we define and observe CNN, which we utilize in the experiment; in finally in Chapter 6, we present the experimental results and discuss possible open research.

Chapter 2

Digital Image Processing

An **image** is a visual representation in the form of a function $f(x, y)$ which represents the color at the point (x, y). Most images are defined over a rectangle. Our prime focus is on **digital images**. These are discrete elements of a 2-D array denoted $f(x, y)$ which we call a pixel representing continuous image $f(x, y)$ that contain rows and columns. Pixels are the smallest units to represent perception of light.The higher the pixel count, the sharper the image. There are three types of images: binary, gray-scale and color images.

2.1 Examples of Binary and Gray-scale Images

Binary images, which are also known to be black and white images, have only two distinct element values for each pixel. If the color is black, we denote the pixel in that position with "0"; If white, we denote the pixel at that position by "1". Gray-level images consists of pixels with various levels of ranging from black to white, $[0, 255]$ respectively. Both binary and gray-scale image arrays are two-dimensional arrays.

Binary images are what we consider a non-dimming light bulb arrays; The light color is on (1) or off (0). Gray-level images are considered dimming because there is a range of grays we could choose from.

(a) Examples of Black and White (Binary).

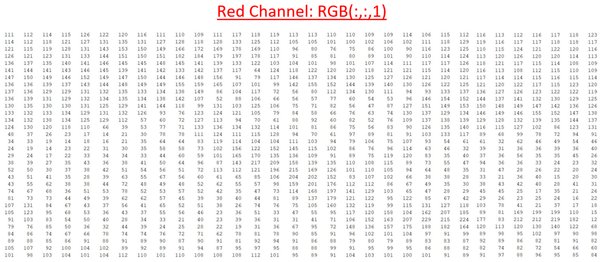
(b) Examples of Gray-scale.

Figure 2.1: Comparison between Binary and Gray-scale images of cats.

2.2 Color Images

Color image pixels are composed of the three primary colors: Red, Green and Blue. Contrary to gray-scale, the color image pixels produce one color by mixing the three primary colors. A color image is arranged as a 3D matrix containing RGB channels in the size of $n_h \times n_w \times 3$ where n_h is the number of pixel rows representing the image's height and n_w is the number of pixel columns of the image representing the width. Here, each of the RGB-channel is 2-D array where each element $R_{i,j}, G_{i,j}, B_{i,j} (i = 1, 2, ..., n_h; j = 1, 2, ..., n_w)$ is an integer between 0 and 255 representing the intensity of primary color at the (i, j)-th position of the image.

As an example, you will find the matrix associated with a $32 \times 32 \times 3$ color cat image as shown below. This image has 3 channels of the primary colors.



Red Channel: RGB(:,:,1)

111	112	114	115	126	122	120	116	111	110	109	111	117	118	119	113	113	110	110	109	109	114	106	115	112	116	113	112	116	117	118	123
114	112	118	121	127	131	135	131	127	128	118	128	133	125	112	105	105	101	100	102	106	102	111	118	129	119	116	117	117	118	118	117
121	115	119	128	131	143	153	150	149	166	172	169	178	169	110	96	80	76	75	86	100	90	116	123	125	110	115	124	121	122	120	116
126	121	123	131	133	144	151	150	151	182	184	179	197	178	117	91	85	81	80	89	101	90	110	114	124	113	120	126	120	120	114	113
136	137	135	140	141	146	145	145	148	145	141	139	133	122	103	104	101	98	101	107	114	111	117	117	126	118	121	117	115	114	108	109
141	144	141	143	146	145	139	141	142	133	142	137	117	64	124	118	122	120	120	118	121	121	115	114	120	116	113	108	112	115	110	109
147	150	149	146	152	149	147	150	144	146	148	156	91	79	117	146	137	134	130	125	127	126	121	120	121	117	114	114	115	116	115	114
136	136	139	137	143	144	148	149	149	155	159	165	107	101	99	142	155	152	144	139	140	130	126	122	125	121	120	122	117	115	123	120
137	136	129	129	131	132	135	133	134	138	149	86	104	117	72	56	80	112	134	130	111	94	93	133	137	136	127	126	123	122	122	119
136	139	131	129	132	134	135	134	138	142	107	52	88	106	66	56	57	77	60	54	53	96	146	154	152	144	137	141	132	130	129	125
130	135	130	130	131	125	129	141	144	118	99	131	103	125	106	75	71	82	56	47	87	127	151	149	153	150	148	149	147	142	136	126
133	132	133	134	129	131	132	126	93	76	123	124	121	105	79	84	58	66	76	63	74	130	137	129	134	146	149	146	155	152	147	138
134	132	138	134	125	129	112	57	60	72	127	113	94	70	61	88	92	60	82	52	76	109	137	138	139	129	128	132	139	135	144	137
124	130	120	118	110	66	39	53	77	71	133	136	134	132	114	101	81	86	75	56	83	90	126	135	140	116	115	127	102	86	123	131
48	37	26	23	17	14	21	30	78	78	111	124	111	115	128	94	70	61	97	89	81	91	103	133	117	89	68	99	78	72	94	91
34	33	19	14	18	16	21	35	64	64	83	119	114	104	104	111	103	94	79	106	75	107	93	54	61	61	32	62	46	49	54	46
24	19	14	23	22	31	30	35	58	58	73	102	156	122	152	145	115	102	86	76	96	114	63	46	32	39	31	36	36	39	36	40
29	24	17	22	33	34	34	33	44	60	59	101	165	170	135	136	109	91	89	75	119	120	83	35	40	37	36	56	35	35	45	26
38	39	27	35	43	36	38	41	50	64	96	87	143	217	209	158	139	135	110	108	115	89	73	55	47	94	36	36	33	24	23	32
52	50	30	37	38	42	51	54	56	51	72	113	112	121	196	215	169	126	101	110	105	94	64	48	35	31	47	28	26	22	20	24
43	51	41	35	28	39	63	55	67	56	60	61	65	85	106	204	202	152	83	107	102	66	38	38	28	33	21	36	40	15	20	30
43	55	62	38	38	44	72	48	49	48	52	62	55	57	98	159	201	176	112	112	86	67	49	35	30	38	43	42	40	28	41	31
74	67	68	36	51	53	78	52	53	57	52	42	35	47	73	114	168	197	141	129	103	65	47	28	29	45	45	25	17	35	21	26
81	73	73	44	49	39	62	62	57	45	39	38	40	44	81	89	137	179	121	122	95	122	85	67	42	29	26	23	25	24	16	22
107	131	84	67	43	37	56	41	65	52	51	38	26	74	76	75	105	160	132	119	99	115	131	127	118	103	78	41	21	37	17	18
105	123	95	68	53	36	43	37	55	56	46	23	36	51	33	47	55	95	117	120	158	104	162	207	185	89	81	169	199	199	110	15
91	103	83	54	50	40	28	34	33	21	40	23	39	36	31	31	41	71	106	152	163	207	229	215	224	177	83	212	212	219	182	12
79	76	85	50	36	32	44	39	24	25	28	22	19	31	36	67	95	72	148	136	157	175	188	182	164	120	113	120	130	140	122	68
84	86	74	67	66	78	74	74	76	72	71	62	78	81	78	90	85	91	96	102	101	104	97	91	99	89	98	95	102	97	90	98
89	88	85	86	91	88	91	89	90	87	90	91	81	92	94	91	86	88	79	80	79	89	83	83	87	92	89	86	82	81	91	82
105	107	92	100	104	102	89	92	89	91	94	87	95	97	95	88	88	99	91	95	99	95	86	88	82	82	74	82	72	54	66	60
101	98	103	104	101	104	112	110	101	110	108	108	108	112	101	95	89	91	104	103	101	100	101	91	89	91	97	88	96	95	85	84

In some applications, such as image segmentation, a color image could be converted into gray-scale images and then a $0 - 1$ mask could be created using appropriate thresholding method to separate for example the foreground from the background of an image. We illustrate this process using the cat image in Figure 2.3 and 0.5 as a simple thresholding value and obtain the following gray-scale image and the corresponding mask. To get better results, more

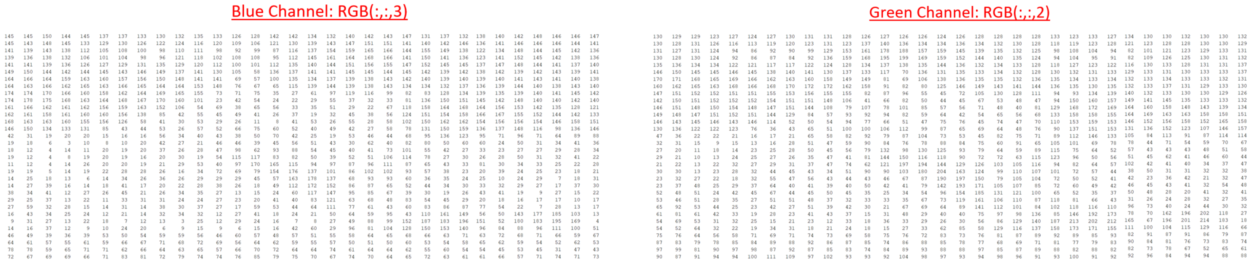

Blue Channel: RGB(:,:,3)
Green Channel: RGB(:,:,2)

Figure 2.3: The Cat Image and associated RGB-Channels of the Color Image.

sophisticated thresholding methods should be used.

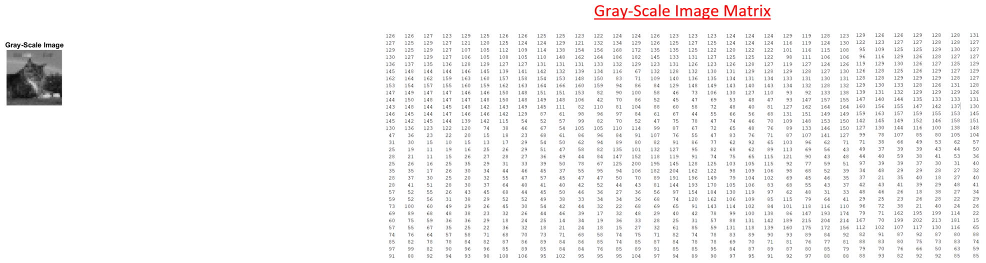


Figure 2.4: The gray-scaled cat image.

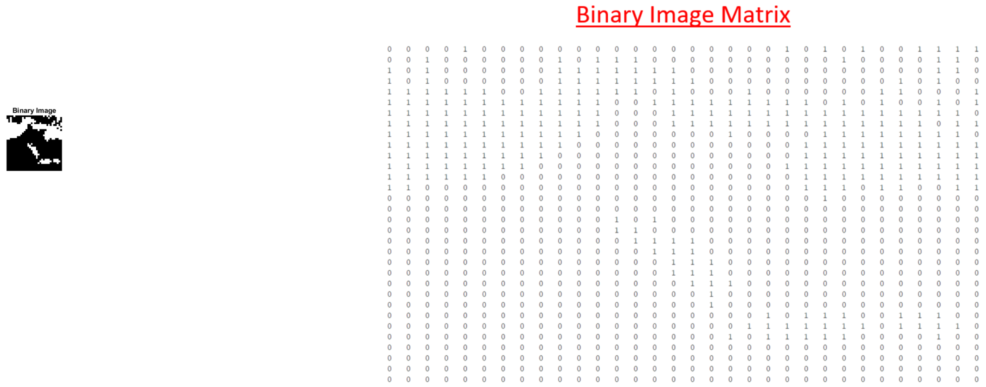


Figure 2.5: The mask of the cat image separating foreground and background.

Chapter 3

Neural Networks With No Hidden Layers

The simplest form of neural networks is one that has no hidden layers and only the output layer and produces a probability between 0 and 1 indicating the likelihood of the input being positive (1) or negative (0) from a linear function of input features. This is equivalent to producing a logistic regression model. The difference is that instead of minimizing a L_2 error function for logistic regression, neural networks minimize the cross-entropy error using gradient descent. In this chapter, we review this process in details.

3.1 Neural Networks without Hidden Layers

A neuron, the basic processing unit of neural networks, processes all fan-in from other nodes (neurons) generating an output with respect to an ***activation function***.

Let's take predicting student's grades of a test as an example. The data set contains six data points.

Single Variable Neural Network. First assume that we only have one input variable x, which is the hours studied for the test. The output variable y is the grade received in the test. Obviously, the hours studied and a grade received are all non-negative quantities. Also, it is common to assume the more a person studies, the higher the grade should be on the test. This leads to a correlation of points in a positive linear direction.

Since we only take one input feature into consideration, the hours studied by students, this could be abstracted as a simple, or single-variable neural network. The model will be

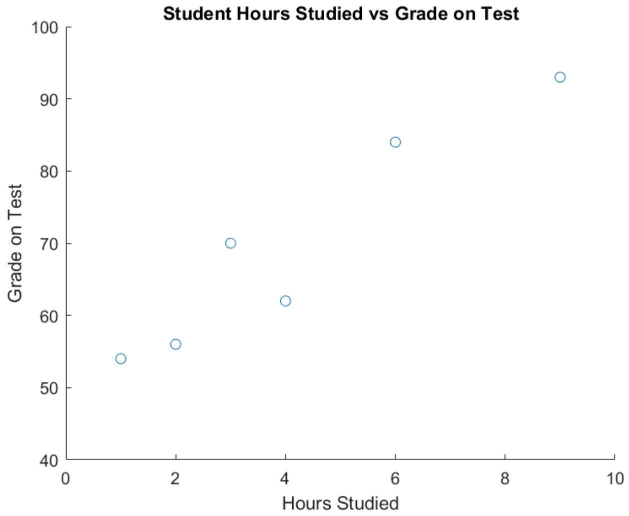


Figure 3.1: Scatterplot of Student Grade Data Set

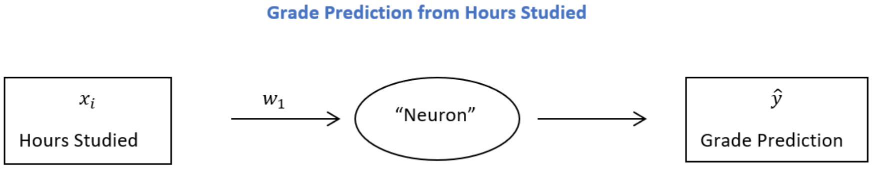


Figure 3.2: A Single Variable Neural Network Without Hidden Layers

represented as shown in Figure 6.

Multi-Variable Neural Networks. Assume that the data set not only includes the input feature of hours studies for the corresponding output price y, but also takes into consideration the number of websites used, the number of hours used on the web for research, and also the number of hours tutored. For this, we have exactly four input variables x_1 representing the hours studied, x_2 representing the number of websites used, x_3 to represent the number of hours used on the web and x_4 to represent the number of hours the students received tutoring and one y output variable, grade. The structure of the ANN could then be abstracted as:

3.2 Linear Regression

For the sake of completeness, we review linear regression and logistic regression in this section. Sir Francis Galton and doctoral student Karl Pearson came across standard deviation in the

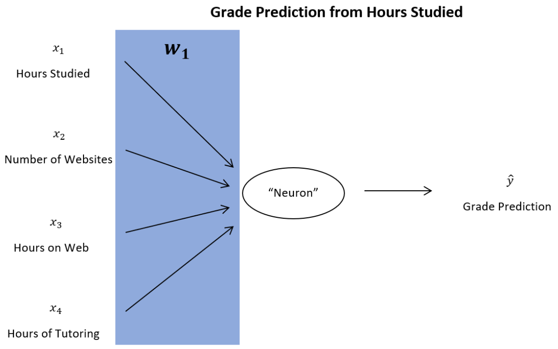


Figure 3.3: A Multiple Input Neural Network Without Hidden Layers

late 1860s. He came up with the concept of correlation. He invented the regression line and was the first to explain the common phenomenon of regression towards the mean. Linear regression, or simply ***regression***, is the comparison between two or more variables. There are two main objectives of linear regression: to establish if there exists a mathematical relationship between two or more variables and to forecast, or find a prediction, potential relationships between unobserved values.

In linear regression, the variable we want to explain, or forecast, is called the *dependent variable* y. The dependent variable is a response based on the independent variable(s) x_i where $i \in \mathbb{N}$. There are two types of linear regression: Univariate and Multiple Linear Regression.

Univariate Regression. In univariate regression we are given a data set with two variables x and y. The goal is to fit the data set to a regression line $y = \beta_0 + \beta_1 x$ that minimizes the distance between the line and the observations. Here, β_1 is the slope, or measure of change between two points and β_0 is the intercept of the y axis estimated based on the data. The slope β_1 measures the sensitivity of the dependent variable on values of the independent values: a positive slope indicates a positive relationship between the dependent and independent variables, while a negative slope indicates a negative relationship.

Multivariate Linear Regression. We can expand the univariate regression model to take into consideration multiple input values for a single output variable. Take into consideration the following data set containing m examples and n input variables:

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

where $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_n^{(i)})$ $(i = 1, 2, ..., m)$ is a n-dimensional vector. We want to fit the data into a regression line:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \cdots \beta_n x_n + \epsilon \tag{3.2.1}$$

where ϵ is a random error following a standard normal distribution.

Least Square Method.

To find the best fitting line, we need to minimize error between the prediction response $\hat{y}^{(i)}$ for each input value and the true output value $y^{(i)}$ using the least square method, one of the contributions by French mathematician Adrien-Marie Legendre. To do so, the steps are outlined below: we first find the initial error $e_i = y_i - \hat{y}_i$. As we can see, the sum of the errors is very small, if not zeros, because large negative error values will offset the positive errors leaving little predictive error to minimize. We can obtain a measured predictive error once we square the original error and take their sum. To be more specific, for a univariate regression model, we define the sum of squared errors (SSE) as:

$$SSE = \sum_{i=1}^{m} e_i^2 = \sum_{i=1}^{m} (y^{(i)} - \beta_0 - \beta_1 x^{(i)})^2 \tag{3.2.2}$$

To find β_0 and β_1 that minimizes SSE, we solve the simultaneous system of equations where $\frac{\partial(SSE)}{\partial\beta_0} = \frac{\partial(SSE)}{\partial\beta_1} = 0$. We observe first that by setting $\frac{\partial(SSE)}{\partial\beta_0} = 0$, $\beta_0 = \overline{Y} - \beta_1 \overline{X}$, where $\overline{Y} = \frac{\sum_{i=1}^{m} y^{(i)}}{m}$ and $\overline{X} = \frac{\sum_{i=1}^{m} x^{(i)}}{m}$ are the mean of all x and y values of our data set.

Proof: First,

$$\frac{\partial(SSE)}{\partial\beta_0} = \sum_{i=1}^{m} -2(y^{(i)} - \beta_0 - \beta_1 x^{(i)})$$

Setting this to 0 yields

$$\sum_{i=1}^{m} y^{(i)} - \beta_1 \sum_{i=1}^{m} x^{(i)} = m\beta_0$$

Dividing both sides by m gives:

$$\beta_0 = \overline{Y} - \beta_1 \overline{X} \ \square$$

Now we take the partial derivative of SSE with respect to β_1:

$$\frac{\partial(SSE)}{\partial\beta_1} = \sum_{i=1}^{m} 2(y^{(i)} - \beta_0 - \beta_1 x^{(i)})(-x^{(i)}) = -2\sum_{i=1}^{m}(y^{(i)}x^{(i)} - \beta_0 x^{(i)} - \beta_1 (x^{(i)})^2)$$

Setting this to 0 yields:

$$\beta_1 \sum_{i=1}^{m}(x^{(i)})^2 = \sum_{i=1}^{m}(y^{(i)}x^{(i)}) - \beta_0 \sum_{i=1}^{m} x^{(i)} \tag{3.2.3}$$

Substituting $\beta_0 = \overline{Y} - \beta_1\overline{X}$ into (3.2.3) and solving for β_1 gives:

$$\beta_1 = \frac{\sum_{i=1}^{m}(y^{(i)}x^{(i)}) - \overline{Y}\sum_{i=1}^{m}(x^{(i)})}{\sum_{i=1}^{m}(x^{(i)})^2 - \overline{X}\sum_{i=1}^{m}(x^{(i)})} \tag{3.2.4}$$

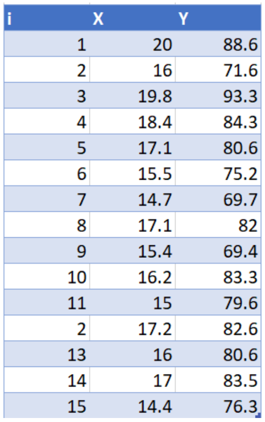
The denominator of (3.2.4) could be rewritten as

$$\sum_{i=1}^{m}(x^{(i)})^2 - \overline{X}\sum_{i=1}^{m}(x^{(i)}) = \sum_{i=1}^{m}[x^{(i)} - \overline{X}]^2$$

For the numerator of (3.2.4), since $\overline{X} = \frac{\sum x^{(i)}}{m}$ and $\overline{Y} = \frac{\sum y^{(i)}}{m}$, it could be rewritten as

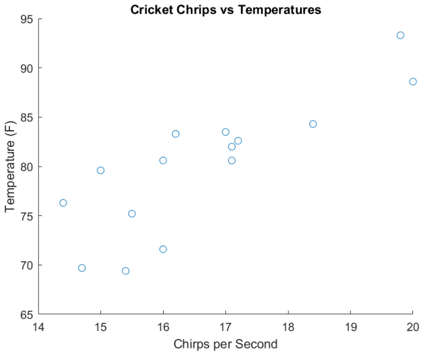
$$\sum_{i=1}^{m}(y^{(i)}x^{(i)}) - \overline{Y}\sum_{i=1}^{m}(x^{(i)}) = \sum_{i=1}^{m}(y^{(i)}x^{(i)}) - \overline{Y}\sum_{i=1}^{m} x^{(i)} + \overline{X}\sum_{i=1}^{m} y^{(i)} - \sum_{i=1}^{m}\overline{XY} = \sum_{i=1}^{m}(y^{(i)} - \overline{Y})(x^{(i)} - \overline{X})$$

Thus, (3.2.4) could be rewritten as



i	X	Y
1	20	88.6
2	16	71.6
3	19.8	93.3
4	18.4	84.3
5	17.1	80.6
6	15.5	75.2
7	14.7	69.7
8	17.1	82
9	15.4	69.4
10	16.2	83.3
11	15	79.6
2	17.2	82.6
13	16	80.6
14	17	83.5
15	14.4	76.3

(a) Data table for associated scatter plot.



(b) Scatterplot

Figure 3.4: Comparison between the temperature (°F) and chirps per second of the stripped ground cricket.

$$\beta_1 = \frac{\sum_{i=1}^{m}(y^{(i)} - \overline{Y})(x^{(i)} - \overline{X})}{\sum_{i=1}^{m}[x^{(i)} - \overline{X}]^2} \tag{3.2.5}$$

(3.2.5) combining with (3.2.4) give a full description of β_0 and β_1 estimated from the data by minimizing SSE.

An Example. As an example, we consider the data set of temperature (°F) (Y) with respect to cricket chirps per second for the stripped ground cricket (X). Below is the scatter plot of the data set.

We want to find a line that will most accurately predict the measure of the temperature (F) given the number of chirps per second of the stripped back cricket. We begin by calculating $\overline{X} = \frac{\sum_{i=1}^{15} x_i}{15} = 16.65$ and $\overline{Y} = \frac{\sum_{i=1}^{15} y_i}{15} = 80.04$. We also find that our slope $\beta_1 = 3.29$. We can now calculate our intercept $\beta_0 = 25.23$ giving us a line of best fit $\hat{y_i} = 25.23 + 3.29x_i$ as shown below

It is clear that there is a positive relation between x and y: as the number of chirps increase, the temperature seems to relatively increase as well.

Coefficient of determination. A common measure for the performance of the linear regression is the coefficient of determination, r^2, defined as the proportion of the variability of

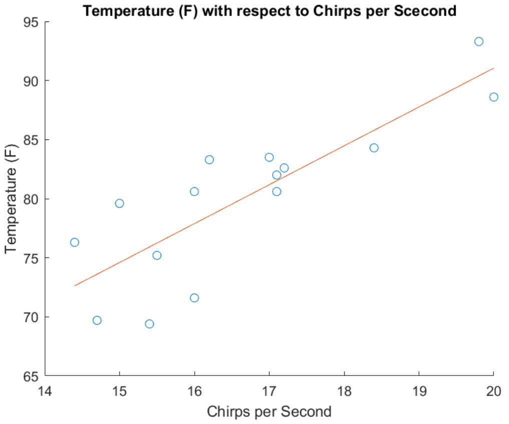


Figure 3.5: Line of Best fit of our data set.

y that is explained by x. This value ranges between 0 and 1 and if $r^2 = 0$ then the line shows no relationship or if $r^2 = 1$ then the prediction shows the relationship with 100% accuracy. It could be calculated using the following formula:

$$r^2 = \frac{SSR}{SST} = \frac{SST - SSE}{SST} = 1 - \frac{SSE}{SST}$$

where $SSR = \sum_{i=1}^{m}(\hat{y}^{(i)} - \overline{Y})^2$ is the the regression sum of squares and $SST = \sum_{i=1}^{m}(y^{(i)} - \overline{Y})^2$ is the total sum of squares. The total sum of variability can be broken into two parts: the first will look at the differences in the height and second takes into consideration other factors that we have not accounted for in the model. This gives us the following formula:

$$SST = SSR + SSE \tag{3.2.6}$$

Let's recall our example of the temperature with respect to the numbers of chirps per second of the stripped back cricket. For our example, we have $SSR = 439.29$. Now we look at the error sum of squares which we defined as SSE in (3.2.4). This leads to $SSE = 190.55$ for our example. Next, we take the total sum of errors SST which gives a value of how much the response varies around their mean. $SST = 629.84$ for our example data set. Finally, we can see that $r^2 = 1 - \frac{SSE}{SST} = 0.566$ for this example.

3.2.1 Logistic Regression

Logistic regression builds upon linear regression and adds a nonlinear link function to transform the output of linear regression, for example, into the interval of [0,1]. A common link function is the sigmoid function:

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

The graph of the sigmoid function is given below:

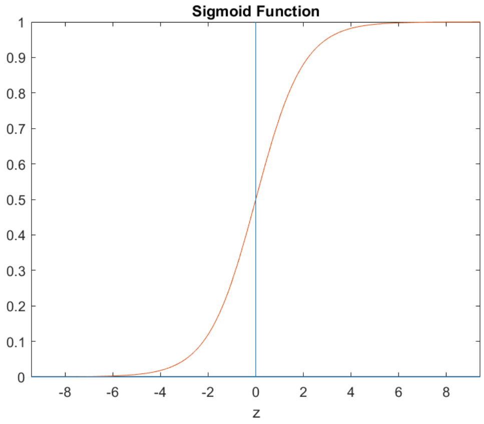


Figure 3.6: Graph of the Logistic Curve/Sigmoid Function

From the graph, it is clear that the sigmoid function has a range between 0 and 1. For logistic regression, the input of the link function $z = \beta_0 + \beta^T\mathbf{x}$ is the output of the linear regression. The output should be interpreted as a posterior probability $\hat{y} = P(y = 1|x)$ of an input data point belonging to the positive (1) class.

3.3 Using Gradient Descent to Train ANN

On contrary to the standard procedure of finding the minimum of squared error, for ANN, we use the gradient descent to find the minimum of the cross entropy error. Recall that from vector calculus, the gradient (or the opposite) of a multi-variable function gives the direction where the value of the function increase (or decrease) the fastest.

To be more specific, for one single training example, as opposed to the linear regression's least square method, which uses the loss function $\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$, we will use the cross entropy error

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

The cross entropy error is inspired by the maximum likelihood estimator in mathematical statistics. We check that it is indeed an "error" function for the following two extreme cases:

- If $y^{(i)} = 1$, then $\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\ln \hat{y}^{(i)}$. This means that we want $-\ln(\hat{y}^{(i)})$ to be as large as possible which means the output $\hat{y}^{(i)}$ must be large (close to 1) as well.
- If $y^{(i)} = 0$, then $\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\ln(1 - \hat{y}^{(i)})$. Here, we want $\ln(1 - \hat{y}^{(i)})$ to be large, which means here, $\hat{y}^{(i)}$ needs to be small (close to 0).

Loss function works for one training example. We must define a cost function that will be able to measure how well the parameters are doing across the entire training data set. The cost function is defined as:

$$\mathcal{J}(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \ln \hat{y}^{(i)} + (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)})] \tag{3.3.1}$$

Our ultimate goal is to find the parameters w and b that minimize the cost function until reaching global optima through an iterative process where at each iteration the parameters (weights) are updated according to the following rule:

$$\theta = \theta - \alpha \nabla \mathcal{J}(\theta)$$

where $\alpha > 0$ is called the **learning rate**.

Each iteration could be broken down into several sub steps:

- Forward propagation: evaluating the cost function using the current value of weights;

- Backward propagation: calculating the gradient of the cost function using chain rule;

- Parameter update: updating the parameters using gradient descent.

3.3.1 Forward Propagation

Given an input matrix $X = \begin{bmatrix} \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \end{bmatrix} \in \mathbb{R}^{n_x \times m}$ in the size of $n_x \times m$, where n_x is the number of features and m is the number of example, the forward propagation process for neural network without hidden layers involves:

First, we multiple the weights and features and then add the bias. Therefore, we obtain $Z = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{bmatrix} = W^T X + b = \begin{bmatrix} W^T x^{(1)} + b \\ W^T x^{(2)} + b \\ \vdots \\ W^T x^{(m)} + b \end{bmatrix} \in \mathbb{R}^{m \times 1}$ as our linear regression model, where $W^T = \begin{bmatrix} w_1 & w_2 & \cdots & w_{n_x} \end{bmatrix} \in \mathbb{R}^{1 \times n_x}$ is the transpose of the weight matrix and $b = \begin{bmatrix} \vdots \\ \vdots \\ b \\ \vdots \\ \vdots \end{bmatrix} \in \mathbb{R}^{m \times 1}$ is an $m \times 1$ matrix by broadcasting the scaler bias m times. We then input Z into the sigmoid activation function which will give us our activation $\hat{Y} = A = \sigma(Z) = \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} = \begin{bmatrix} \sigma(z^{(1)}) \\ \sigma(z^{(2)}) \\ \vdots \\ \sigma(z^{(m)}) \end{bmatrix} \in \mathbb{R}^{1 \times m}$. Finally, we calculate the cost function $\mathcal{J}(W, b)$.

This process is illustrated in the following figure for an input matrix that has 3072 features

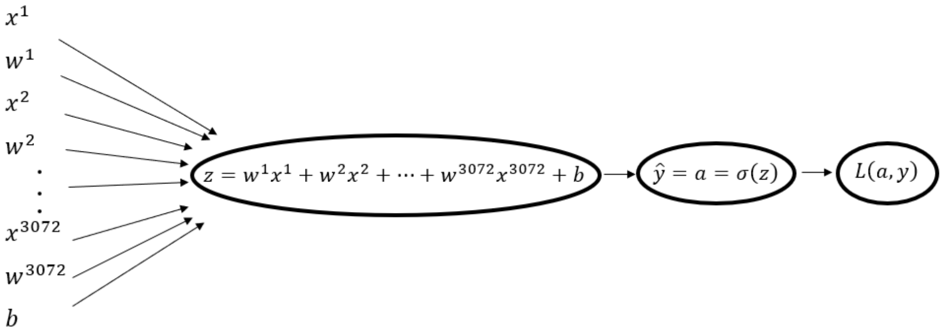

x^1
w^1
x^2
w^2
x^{3072}
w^{3072}
b
$z = w^1x^1 + w^2x^2 + \cdots + w^{3072}x^{3072} + b$
$\hat{y} = a = \sigma(z)$
$L(a, y)$

Figure 3.7: Forward propagation of a single example.

and 1 example:

3.3.2 Backward Propagation

The next step is calculating the gradient of the cost function through a propagation process that starts backward from the cost function.

We illustrate this process using one example first: given an input matrix and parameters W and b, recall

$$z = W^T x + b$$

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

$$\mathcal{L}(a, y) = -(y \ln a + (1 - y) \ln(1 - a))$$

The parameters needs to be updated to minimize error; therefore, we need to take the partial derivatives of the Loss function with respect to a first, as shown below:

$$\begin{aligned} \text{“}da\text{”} = \frac{\partial \mathcal{L}(a, y)}{\partial a} &= \frac{\partial[-(y \ln a + (1 - y) \ln(1 - a))]}{\partial a} \\ &= -\left(\frac{y}{a} + (1 - y)\frac{-1}{1 - a}\right) \\ &= -\frac{y}{a} + \frac{1 - y}{1 - a} \end{aligned}$$

We then take the partial derivative of $\mathcal{L}(a, y)$ with respect to z giving:

$$\begin{aligned}
\text{“}dz\text{”} = \frac{\partial \mathcal{L}(a, y)}{\partial z} &= \frac{\partial[-(y \ln a + (1-y)\ln(1-a))]}{\partial z} \\
&= \frac{\partial\left[-\left(y \ln\left(\frac{1}{1+e^{-z}}\right) + (1-y)\ln\left(1-\left(\frac{1}{1+e^{-z}}\right)\right)\right)\right]}{\partial z} \\
&= \frac{\partial[-(y \ln((1+e^{-z})^{-1}) + (1-y)\ln(1-(1+e^{-z})^{-1}))]}{\partial z} \\
&= -\Bigg[y\left(\frac{1}{(1+e^{-z})^{-1}}\right) - (1+e^{-z})^{-2}(-e^{-z}) + (1-y)\left(\frac{1}{1-(1+e^{-z})^{-1}}\right)(1+e^{-z})^{-2}(-e^{-} \\
&= -\left(\frac{y(1+e^{-z})(e^{-z})}{(1+e^{-z})^2} - \frac{(1-y)(1+e^{-z})(e^{-z})}{e^{-z}(1+e^{-z})^2}\right) \\
&= -\frac{ye^{-z} + (1-y)}{1+e^{-z}} \\
&= \frac{1 - y - ye^{-z}}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}} - \frac{y(1+e^{-z})}{1+e^{-z}} \\
&= a - y
\end{aligned}$$

Another approach to solving for dz is by taking the product of da and $\frac{da}{dz}$. From above, we know that $da = -\frac{y}{a} + \frac{1-y}{1-a}$ and given $a = \frac{1}{1+e^{-z}} = (1+e^{-z})^{-1}$

$$\begin{aligned}
\frac{da}{dz} &= \frac{d(1+e^{-z})^{-1}}{dz} \\
&= -(1+e^{-z})^{-2}(-e^{-z}) \\
&= \frac{-e^{-z}}{(1+e^{-z})^2} \\
&= \frac{-e^{-z} + 1 - 1}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2} \\
&= (a - a^2) \\
&= a(1-a)
\end{aligned}$$

So,

$$
\begin{aligned}
\text{``}dz\text{''} = \frac{\partial \mathcal{L}(a,y)}{\partial z} &= \frac{\partial \mathcal{L}(a,y)}{\partial a} \cdot \frac{da}{dz} \\
&= -\frac{y}{a} + \frac{1-y}{1-a} \cdot a(1-a) \\
&= \frac{-y[a(1-a)]}{a} + \frac{(1-y)[a(1-a)]}{1-a} \\
&= \frac{-y[a(1-a)(1-a)] + (1-y)[a^2(1-a)]}{a(1-a)} \\
&= \frac{a(-y + 2ya - ya^2 + a - ya - a^2 + ya^2)}{a(1-a)} \\
&= \frac{-y + 2ya - ya^2 + a - ya - a^2 + ya^2}{1-a} \\
&= \frac{-y + ya + a - a^2}{1-a} \\
&= \frac{(1-a)(-y+a)}{1-a} \\
&= -y + a \\
&= a - y
\end{aligned}
$$

We finally find the partial derivative of $\mathcal{L}$ with respect to the weights and bias. By chain rule:

$$
\begin{aligned}
\text{``}dw\text{''} = \frac{\partial \mathcal{L}(a,y)}{\partial w} &= dz \cdot \frac{dz}{dw} \\
&= (a-y)\frac{d(W^T x + b)}{dw} \\
&= (a-y)(x) = (x)dz
\end{aligned}
$$

$$\begin{aligned}\text{“}db\text{”} = \frac{\partial \mathcal{L}(a,y)}{\partial b} &= dz \cdot \frac{dz}{db} \\ &= (a-y)\frac{d(W^T x + b)}{db} \\ &= (a-y)(b) = (b)dz\end{aligned}$$

We can expand backward propagation across multiple training examples as well. We take the gradient of $\mathcal{J}(W, b)$ with respect to parameters W and b. We find that for one example, $dz = a - y$. Therefore, the multiple example version of the formula is given as

$$dZ = A - Y$$

where $A = \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times 1}$ is the activation and $Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}$ is the given output matrix.

We now take the partial derivative of the cost function with respect to the parameters. Let's first take

$$db = \frac{\partial J(W,b)}{\partial b} = \frac{\partial(\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}))}{\partial b} = \frac{1}{m}\frac{\sum_{i=1}^{m} \partial \mathcal{L}(a^{(i)}, y^{(i)})}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) = \overline{A} - \overline{Y}$$

We now take

$$dW = \frac{\partial J(W,b)}{\partial W} = \frac{\partial \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})}{\partial W} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}(a^{(i)}, y^{(i)})}{\partial Z}\frac{\partial Z}{\partial W} = \frac{1}{m}dZ\sum_{i=1}^{m}(x^{(i)}) = \frac{1}{m}XdZ = \overline{X}dZ$$

3.3.3 Parameter Update

We now update the parameters in the direction of the derivative by

$$\begin{cases} W = W - \alpha dW \\ \\ b = b - \alpha db \end{cases} \tag{3.3.2}$$

This process is repeated for a certain number of times that we predetermine.

3.3.4 Performance Evaluation

Once the neural network is trained, we will use the trained parameters to make predictions using cut-off probability of 0.5. If the activation obtained using the trained parameter for example i, $a^{(i)} \geq 0.5$ then we will predict $\hat{y}^{(i)} = 1$; otherwise, $\hat{y}^{(i)} = 0$. To check the performance of the algorithm, we check the accuracy on both the training and testing sets calculated by

$$\begin{cases} Accuracy_{train} = 1 - \dfrac{1}{m} \displaystyle\sum_{i=1}^{m} \left| \hat{y}_{train}^{(i)} - y_{train}^{(i)} \right| \\ Accuracy_{test} = 1 - \dfrac{1}{m} \displaystyle\sum_{i=1}^{m} \left| \hat{y}_{test}^{(i)} - y_{test}^{(i)} \right| \end{cases} \tag{3.3.3}$$

Chapter 4

Neural Networks with Hidden Layers

A weakness of using a simple logistic regression model or neural network without any hidden layers mentioned in the chapter is that they are not sufficient to model the complicated relations in real life accurately. Therefore, we review more sophisticated neural networks with hidden layers. We will call a neural network with exactly one hidden layer to be a shallow neural network formed by adding hidden layer of multiple neurons between the input layer and output layer.(Ng) At each neuron, there are two actions that are performed, the linear transform followed by a nonlinear activation function. From there, the activation function's output is then sent to the next hidden layer or the output layer. The following figure shows the structure of a shallow neural network with a four-neuron hidden layer:

4.1 Common Activation Functions

A critical difference between ANN without hidden layers and with hidden layers is that the activation function for the hidden layers is usually not the sigmoid function. We review some of the most common activation functions for hidden layers. So far, we have discussed only the sigmoid function $g(z) = \frac{1}{1+e^{-z}}$ as an activation function. There are other activation functions that we can utilize. We have to take the derivative the activation function with respect to the linear regression model in our model for gradient descent. The derivative of the Sigmoid function is given as $\frac{dg(z)}{dz} = g(z)(1 - g(z))$.

Another activation function we can observe is the Hyperbolic Tangent ($tanh$) function which has an output range of $[-1, 1]$ from any input value. The Logistic Sigmoid function can cause

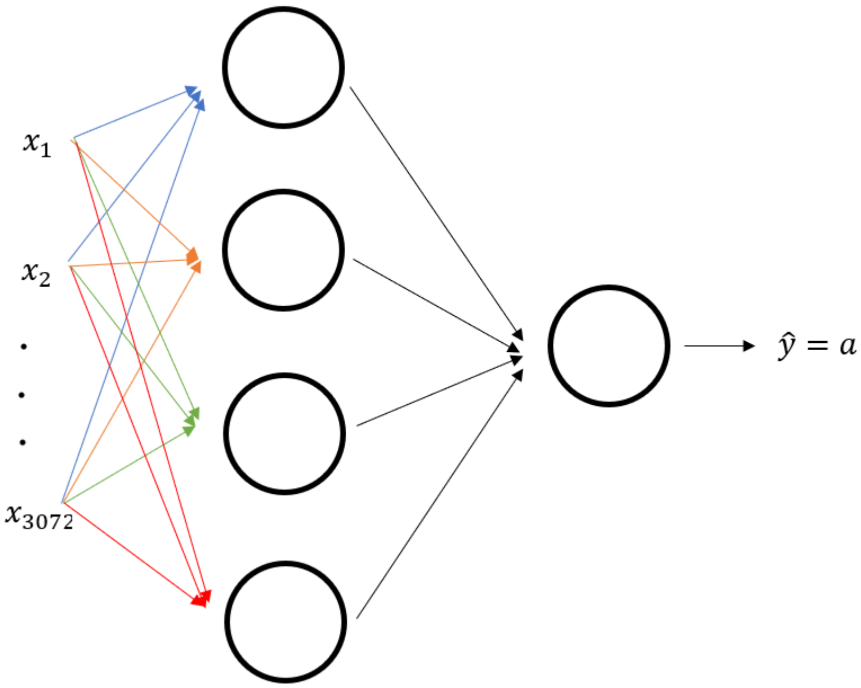


Figure 4.1: Shallow Neural Network of a single example.

the neural network to be ”stuck”, or barely change. A very large, or strong, negative number would cause an output very close to zero. Our objective to to utilize forward propagation to then utilize gradient descent on the parameters to calculate updated parameters. If near zero, there would be seemingly little to no change in the algorithm. Therefore, in a neural network, we will only utilize the Logistic Sigmoid function in the last layer, or Output layer. for the hidden layers, we have a few options to chose between.

We can utilize a smoother activation function, as opposed to the sigmoid function, the Hyperbolic Tangent function. This allows data with strong negative values map to negative output values. This will allow the network to operate more efficiently. Below, you will find the associated graph for the Hyperbolic Tangent activation function, $g_{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$.

We take the derivative of $g_{tanh}(z)$ with respect to z during gradient descent as outlined below:

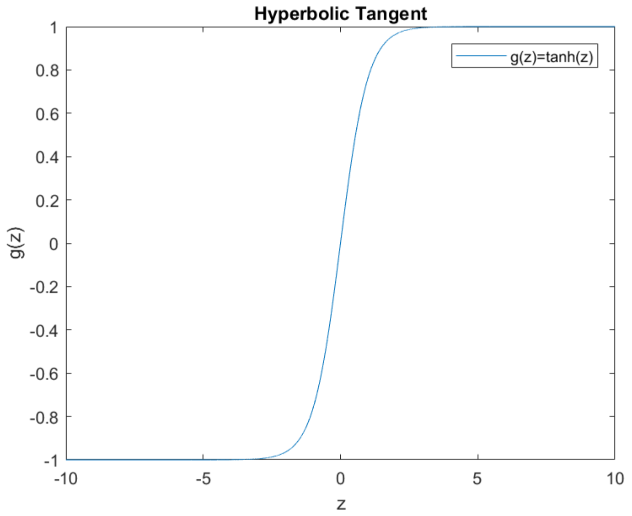


Figure 4.2: Hyperbolic Tangent activation function.

$$
\begin{aligned}
g_{tanh}(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\
&= (e^z - e^{-z})(e^z + e^{-z})^{-1} \\
\frac{dg_{tanh}(z)}{dz} &= (e^z - e^{-z})[-(e^z + e^{-z})^{-2}(e^z - e^{-z})] + (e^z + e^{-z})(e^z + e^{-z})^{-1} \\
&= -(e^z - e^{-z})^2(e^z + e^{-z})^{-2} + 1 \\
&= -\frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2} + 1 \\
&= -\left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2 + 1 \\
&= 1 - g_{tanh}(z)^2
\end{aligned}
$$

We now take a look at the most widely used activation function in the world, the Rectifier (ReLU) activation function. This function defines only the positive values of $z^{[i]}$. This leads to the output value range between $[0, \infty)$.

The issue with the ReLU activation function is that negative values input into the function are all mapped to 0, decreasing the ability the train the data appropriately. The issue is that the negative values of the data set are not being reflected on the training model at all. There

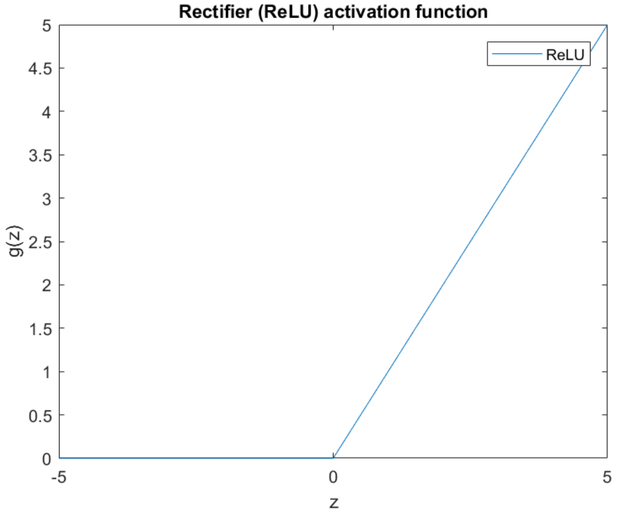


Figure 4.3: Rectifier (ReLU) activation function.

are instances where the ReLU model is valuable, therefore we define the Rectifier function and its derivative below:

$$g_{ReLU}(z) = max(0, z) = \begin{cases} 0 & z \leq 0 \\ z & z > 0 \end{cases} \tag{4.1.1}$$

Although the derivative of the ReLU function at 0 is not defined, the derivative is well defined elsewhere. We could simply define the derivative of the ReLU function at 0 to be 1 and obtain

$$g'_{ReLU}(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{4.1.2}$$

A variant of the ReLU function, the leaky ReLU function is also commonly used and is defined below:

$$g_{LeakyReLU}(z) = max(0.01z, z) = \begin{cases} 0.01z & z \leq 0 \\ z & z > 0 \end{cases} \tag{4.1.3}$$

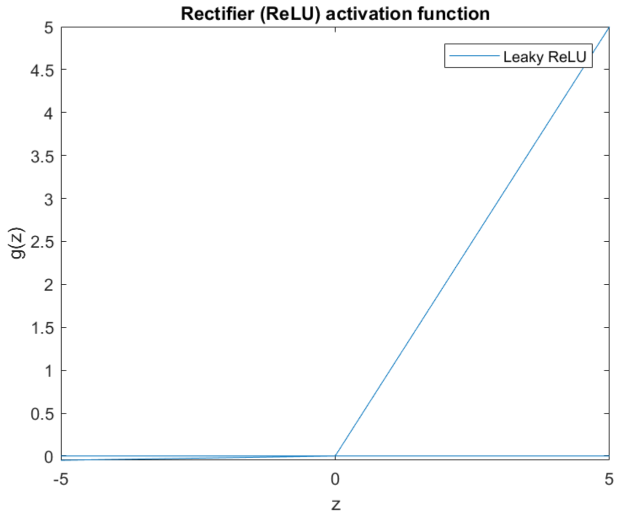


Figure 4.4: Leaky Rectifier (ReLU) activation function.

Similarly, the leaky ReLU function has the derivative defined below

$$g'_{LeakyReLU}(z) = \begin{cases} 0.01 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{4.1.4}$$

4.2 Gradient Decent Optimization

To find the global minimum of the cost function for neural networks with hidden layers, we also use the gradient descent algorithm.

4.2.1 Forward Propagation

Let $X = [x^{(1)}|x^{(2)}|...|x^{(m)}]$, an $n_x \times m$ matrix be the input data of m examples, $Z^{[l]} = [z^{[l](1)}|z^{[l](2)}|...|z^{[l](m)}]$ be the linear transform of X at layer l and $A^{[l]} = [a^{[l](1)}|a^{[l](2)}|...|a^{[l](m)}]$ be the activation matrix of layer l. In addition, assume that (hidden) layer l has $n^{[l]}$ number of neurons, $l = 0, ..., L$. Then, the 0-th layer is the input layer while the L-th layer is the output layer. Thus, $n^{[0]} = n_x$, the number of features and $n^{[L]} = 1$, since the output layer only has one neuron.

The forward propagation for m examples is given by:

- At Hidden Layer 1: $Z^{[1]} = W^{[1]}X + b^{[1]}$;

  $A^{[1]} = g^{[1]}(Z^{[1]})$

  $\vdots$

- At Hidden Layer l: $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$

  $A^{[l]} = g^{[l]}(Z^{[l]})$

  $\vdots$

- At Output Layer, Layer L: $Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$

  $A^{[L]} = g^{[L]}(Z^{[L]})$

where $W^{[1]}$ is the (transposed) weight matrix for layer 1 in the size of $n^{[1]} \times n_x$, $Z^{[1]}$ is the linear transformation of X at layer 1 in the size of $n^{[1]} \times 1$ and $A^{[1]}$ is the activation matrix of layer 1 in the size of $n^{[1]} \times m$. Moving forward for layer l we have a weight matrix (transposed) $W^{[l]}$ in the size of $n^{[l]} \times n^{[l-1]}$, the linear transformation of $A^{[l-1]}$ at layer l is $Z^{[l]}$ in the size of $n^{[1]} \times m$ and then the activation matrix of layer l is $A^{[l]}$ in the size of $n^{[1]} \times m$. We continue to move through the ANN until we reach the final Output layer L where $W^{[L]}$ is the weight matrix (transposed) for layer L in the size of $n^{[L]} \times n^{[L-1]}$. $Z^{[L]}$ is the linear transformation of $A^{[L-1]}$ at layer L in the size of $n^{[L]} \times m$ which leads to the activation matrix of layer L in the size of $n^{[L]} \times m$.

4.2.2 Backward Propagation

Once we complete the forward propagation, we apply backward propagation to find the gradient of the cost function. The backward propagation process are summarized by the following formula: for output layer, Layer L:

- $dZ^{[L]} = A^{[L]} - Y$ of size $n^{[L]} \times m$.
- $dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]^T}$ of size $n^{[L]} \times n^{[L-1]}$.

- $db^{[L]} = \frac{1}{m} dZ^{[l]} \mathbf{1}$ of size $n^{[L]} \times 1$.

For all other layers, Layer l, $(l = 0, 1, ..., L-1)$,

- $dZ^{[l]} = W^{[l+1]^T} dZ^{[l+1]} * g^{[l]'}(Z^{[l]})$ of size $n^{[l]} \times m$.
- $dW^{[l]} = \frac{1}{m} dZ^{[l]} X^T$ of size $n^{[l]} \times n^{[l-1]}$.
- $db^{[l]} = \frac{1}{m} dZ^{[l]} \mathbf{1}$ of size $n^{[l]} \times 1$.

Here, $*$ is the element-wise multiplication and $\mathbf{1}$ is a column matrix of identically one's in the size of $m \times 1$.

4.2.3 An Example

We illustrate the process using a shallow neural network that has $n_x = 3$ features and $n^{[1]} = 4$ nodes in the first hidden layer trained using one example. To begin process, we initialize the set of weights and bias through the first layer of the neural network giving

$$W^{[1]} = \begin{bmatrix} w_{1,1}^{[1]} & w_{2,1}^{[1]} & w_{3,1}^{[1]} \\ w_{1,2}^{[1]} & w_{2,2}^{[1]} & w_{3,2}^{[1]} \\ w_{1,3}^{[1]} & w_{2,3}^{[1]} & w_{3,3}^{[1]} \\ w_{1,4}^{[1]} & w_{2,4}^{[1]} & w_{3,4}^{[1]} \end{bmatrix} \in \mathbb{R}^{4\times 3}$$

and

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \in \mathbb{R}^{4\times 1}$$

which are then fed into the linear regression model

$z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[1]}x_1 + w_{2,1}^{[1]}x_2 + w_{3,1}^{[1]}x_3 + b_1^{[1]} \\ w_{1,2}^{[1]}x_1 + w_{2,2}^{[1]}x_2 + w_{3,2}^{[1]}x_3 + b_2^{[1]} \\ w_{1,3}^{[1]}x_1 + w_{2,3}^{[1]}x_2 + w_{3,3}^{[1]}x_3 + b_3^{[1]} \\ w_{1,4}^{[1]}x_1 + w_{2,4}^{[1]}x_2 + w_{3,4}^{[1]}x_3 + b_4^{[1]} \end{bmatrix} \in \mathbb{R}^{4\times 1}$. We now utilize the activation function obtaining $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \begin{bmatrix} g_1^{[1]}(z_1^{[1]}) \\ g_2^{[1]}(z_2^{[1]}) \\ g_3^{[1]}(z_3^{[1]}) \\ g_4^{[1]}(z_4^{[1]}) \end{bmatrix} = g(z^{[1]}) \in \mathbb{R}^{4\times 1}$ where $g(\star)$ is the activation function chosen through Layer 1. We now initialize out second set of weights

$$W^{[2]} = \begin{bmatrix} w_1^{[2]} & w_2^{[2]} & w_3^{[2]} & w_4^{[2]} \end{bmatrix}$$

of size 1 × 4 and bias $b^{[2]}$ that will pass through the final output layer giving $z^{[2]} = w_1^{[2]}a_1^{[1]} + w_2^{[2]}a_2^{[1]} + w_3^{[2]}a_3^{[1]} + w_4^{[2]}a_4^{[1]} + b^{[2]}$. We now implement our final activation function, which would be the sigmoid function for binary classification giving $a^{[2]} = g(z^{[2]})$.

Next, we calculate the partial derivative of the loss function (since we only have one training example). The first one is

$$dz^{[2]} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} = a^{[2]} - y$$

For $dw^{[2]} = \frac{\partial \mathcal{L}}{\partial w^{[2]}}$, we have

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial w_2^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial w_3^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial w_4^{[2]}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w_1^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w_2^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w_3^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w_4^{[2]}} \end{bmatrix} = \begin{bmatrix} (a^{[2]} - y)a_1^{[1]} \\ (a^{[2]} - y)a_2^{[1]} \\ (a^{[2]} - y)a_3^{[1]} \\ (a^{[2]} - y)a_4^{[1]} \end{bmatrix} = dz^{[2]}a^{[1]T}$$

We continue the backward propagation by finding

$$dz^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} = \begin{bmatrix} \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dz_1^{[1]}} \\ \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dz_2^{[1]}} \\ \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dz_3^{[1]}} \\ \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dz_4^{[1]}} \end{bmatrix} = \begin{bmatrix} dz^{[2]} w_1^{[2]} g^{[1]\prime}(z_1^{[1]}) \\ dz^{[2]} w_2^{[2]} g^{[1]\prime}(z_2^{[1]}) \\ dz^{[2]} w_3^{[2]} g^{[1]\prime}(z_3^{[1]}) \\ dz^{[2]} w_4^{[2]} g^{[1]\prime}(z_4^{[1]}) \end{bmatrix} = w^{[2]T} dz^{[2]} . * g^{[1]\prime}(z^{[1]})$$

where $*$ is the element-wise multiplication.

We continue by finding the updated parameters for the first set of weights obtaining

$$\begin{aligned} dw^{[1]} = \frac{\partial \mathcal{L}}{\partial w^{[1]}} &= \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{1,1}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{1,2}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{1,3}^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{2,1}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{2,2}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{2,3}^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{3,1}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{3,2}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{3,3}^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{4,1}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{4,2}^{[1]}} & \frac{\partial \mathcal{L}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{4,3}^{[1]}} \end{bmatrix} \\ &= \begin{bmatrix} dz_1^{[1]} a_1^{[0]} & dz_1^{[1]} a_2^{[0]} & dz_1^{[1]} a_3^{[0]} \\ dz_2^{[1]} a_1^{[0]} & dz_2^{[1]} a_2^{[0]} & dz_2^{[1]} a_3^{[0]} \\ dz_3^{[1]} a_1^{[0]} & dz_3^{[1]} a_2^{[0]} & dz_3^{[1]} a_3^{[0]} \\ dz_4^{[1]} a_1^{[0]} & dz_4^{[1]} a_2^{[0]} & dz_4^{[1]} a_3^{[0]} \end{bmatrix} \\ &= \begin{bmatrix} dz_1^{[1]} \\ dz_2^{[1]} \\ dz_3^{[1]} \\ dz_4^{[1]} \end{bmatrix} \begin{bmatrix} a_1^{[0]} & a_2^{[0]} & a_3^{[0]} \end{bmatrix} = dz^{[1]T} X \end{aligned}$$

4.2.4 Random Initialization

With logistic regression the parameters could be initialized with zeros and still function. If we let the weights for all layers in an ANN be zeros, we will notice that as we move forward, $A^{[l-1]} = A^{[l]}$ and during backward propagation, we find that $dZ^{[l-1]} = dZ^{[l]}$, which will give us the exact same algorithm causing symmetry between the the layers. We can allow the bias to be initialized as zeros since there is not a massive change or impact on the network. To avoid this, we have to randomly initialize the weight using the Gaussian random variable the size of the weight matrix and multiply it by an very small number as shown:

$$W^{[l]} = \mathbf{S} * 0.001, \ l = 1, 2, ..., L$$

where $\mathbf{S} = \{s_{i,j}\}$ is a matrix in the size of $n^{[l]}$ by $n^{[l-1]}$ with each entry $s_{i,j}$ being a standard normal random variable. If we do not multiply a very small number, we will find that as we move forward through the ANN, the linear transformation would be very large. Since we have a binary classification problem, we utilize the Sigmoid function as the activation function in the Output layer. We will notice that if our linear transformation is large, then the activation of it will be very large, which will cause the gradient descent to be very slow which is not good productivity wise.

Chapter 5

Convolutional Neural Network: CNN

Convolutional Neural Network (CNN) extends the traditional ANN by adding the operations of convolution and pooling (subsampling).(Ng) Thus, CNNs are able to detect important features, such as edges, shapes and patterns in digital images more efficiently than a traditional ANN. In recent years, CNNs have quickly developed into a field of computer vision. Applications of CNNs include self-driving cars, face recognition and even modern artwork. Some type problems CNN help solve include image classification, object detection and neural style transfer, taking two images where we combine features to create another image. In this chapter, we review some basic operations of CNN and the architecture of some important CNNs that will be applied later to the binary classification problem.

5.1 Basic Operations of CNN

Given m training examples of input image matrices $X^{(m)} = a^{[0](m)}$ which has dimensions $m \times n_H^{[0](m)} \times n_W^{[0](m)} \times n_c^{[0]}$, where $n_H^{[0](m)}$ represents the the height of training example m image matrix of layer [0], $n_W^{[0](m)}$ represents the the width of training example m image matrix of layer [0], and $n_c^{[0](m)}$ represent the number of channels of training example m image matrix of layer [0]. Our goal is to send a set of image matrices through a Convolutional Neural Network to obtain an activation function that will minimize the error between the actual and predicted outcome.

The weights we send the image matrix through is considered a filter, which we can predetermine or let the network train after initializing. We do predetermine the size of our filter; Every matrix must has it own filter matrix, therefore, we can predetermine our filter to have dimension

$f_H^{[1](m)} \times f_W^{[1](m)} \times n_c^{[0]} \times n_c^{[1]}$ where $f_H^{[1](m)}$ and $f_W^{[1](m)}$ represent the height and width of the filter for training example m in layer [1], respectively, $n_c^{[0]}$ represent the number of channels that must be present to operate, therefore, we find that the this will always be equal to the number of channels of the input image, and finally $n_c^{[1]}$ represents the total number of filters that must be present since there could be more than one training example.

5.1.1 Convolution

Given a 3D matrix $A = \{a_{i,j,k}\}$ in the size of $n_h^A \times n_w^A \times n_c^A$ and a 3D filter matrix $B = \{b_{l,m,n}\}$ in the size of $n_h^B \times n_w^B \times n_c^B$ where $n_h^B \leq n_h^A, n_w^B \leq n_w^A, n_c^B = n_c^A$, the **convolution** matrix $C = \{c_{s,t}\} = A * B$ of A and B is a matrix in the size of $(n_h^A - n_h^B + 1) \times (n_w^A - n_w^B + 1)$ where

$$c_{s,t} = \sum_{k=1}^{n_c^A} \sum_{j=t}^{t+n_w^B-1} \sum_{i=s}^{s+n_h^B-1} a_{i,j,k} b_{i-s+1,j-t+1,k}$$

As an example, the convolution of an input gray-scale (2D) 8×8 image and a filter of size 3 is shown below:

$$\begin{bmatrix} 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & -1 & -7 & -1 & 7 & 2 \\ 5 & -1 & -7 & -1 & 7 & 2 \\ -5 & -3 & 8 & -11 & -3 & 12 \\ -5 & 9 & -7 & -11 & 7 & 2 \\ 5 & -1 & -7 & -1 & 7 & 2 \\ 5 & -1 & -7 & -1 & 7 & 2 \end{bmatrix}$$

The motivation of doing convolution is that by applying appropriate filters, it could detect various features in an image. For example, the filter above is a *vertical edge detector*. Another example is a horizontal edge detector:

$$\begin{bmatrix} 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -4 & -4 & -4 & -4 & 6 & 6 \\ 2 & 2 & 2 & 2 & -8 & -8 \\ 4 & 4 & 4 & 4 & -6 & -6 \\ -2 & -2 & -2 & -2 & 8 & 8 \\ -4 & -4 & -4 & -4 & 6 & 6 \\ 2 & 2 & 2 & 2 & -8 & -8 \end{bmatrix}$$

The convolution operation replace the linear transformation in a traditional ANN and the output of the convolution is then sent to a nonlinear activation function (mentioned in the previous chapters). Some technical treatments, such as striding and padding, could be applied to convolution; to for example, keep the height and width of the input image matrix.

Stride. Given a 3D matrix $A = \{a_{i,j,k}\}$ in the size of $n_h^A \times n_w^A \times n_c^A$ and a 3D filter matrix $B = \{b_{l,m,n}\}$ in the size of $n_h^B \times n_w^B \times n_c^B$ where $n_h^B \leq n_h^A, n_w^B \leq n_w^A, n_c^B = n_c^A$, the **convolution** matrix $C = \{c_{s,t}\} = A * B$ of A and B **with a stride** n_s is a matrix in the size of $\lfloor \frac{n_h^A - n_h^B}{n_s} + 1 \rfloor \times \lfloor \frac{n_w^A - n_w^B}{n_s} + 1 \rfloor$ where

$$c_{s,t} = \sum_{k=1}^{n_c^A} \sum_{j=tn_w^B-1}^{(tn_w^B-1)+n_w^B-1} \sum_{i=sn_h^B-1}^{(sn_h^B-1)+n_h^B-1} a_{i,j,k} b_{i-s+1,j-t+1,k}$$

Essentially, striding means we apply the filter to the input image matrix by skipping every $n_s - 1$ element in the horizontal direction. As an example, the convolution of the input gray-scale (2D) 8×8 image used in the previous subsection and the same filter of size 3 *with stride 3* is shown below:

$$\begin{bmatrix} 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & -1 \\ -5 & -11 \end{bmatrix}$$

The motivation of doing convolution is by applying appropriate filters with stride, it could detect various features in an image. For example, the filter above is a *vertical edge detector* with stride 3. Another example is a horizontal edge detector:

$$\begin{bmatrix} 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -4 & -4 \\ -2 & -2 \end{bmatrix}$$

Notice that with the convolution and stride, the Output matrix size is smaller than the Input matrix. Also, there are times when we stride, all data of the Input image is not taken into consideration, as shown above. That data not taken into consideration may be vital information that is not being processed in the ANN. To prevent such things, we utilized the concept of padding. **Padding.** Given a 3D matrix $A = \{a_{i,j,k}\}$ with padding n_p of the size of $n_h^A + 2n_p \times n_w^A + 2n_p \times n_c^A$ and a 3D filter matrix $B = \{b_{l,m,n}\}$ in the size of $n_h^B \times n_w^B \times n_c^B$ where

$n_h^B \leq n_h^A, n_w^B \leq n_w^A, n_c^B = n_c^A$, the **convolution** matrix $C = \{c_{s,t}\} = A * B$ of A and B with a stride n_s gives a matrix in the size of $\lfloor \frac{(n_h^A+2n_p)-n_h^B}{n_s} + 1 \rfloor \times \lfloor \frac{(n_w^A+2n_p)-n_w^B}{n_s} + 1 \rfloor$ where

$$c_{s,t} = \sum_{k=1}^{n_c^A} \sum_{j=tn_w^B-1}^{(tn_w^B-1)+n_w^B-1} \sum_{i=sn_h^B-1}^{(sn_h^B-1)+n_h^B-1} a_{i,j,k} b_{i-s+1,j-t+1,k}$$

Essentially padding is the addition of equal number of columns and rows of zeros n_p where the dimensions of the Input image change to $n_h^A + 2n_p$ and $n_w^A + 2n_p$.

The amount of padding is predetermined before performing the convolution operation. It is common practice to have the size of the Output matrix to be the same size as the size of the Input matrix. To figure out n_p, we set the size of the Output matrix with padding equal to the size of A before padding then evaluate for n_p as shown below

$$\begin{aligned}
\frac{(n^A + 2n_p) - n^B}{n_s} + 1 &= n^A \\
(n^A + 2n_p) - n^B &= (n^A - 1)n_s \\
2n_p &= (n^A - 1)n_s + n^B - n^A \\
n_p &= \frac{(n^A - 1)n_s + n^B - n^A}{2}
\end{aligned}$$

As an example, the convolution of the input gray-scale (2D) 8×8 image used in the previous subsection and the same filter of size 3 *with stride 3 and padding of 1* is shown below:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 & 0 \\ 0 & 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 & 0 \\ 0 & 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 & 0 \\ 0 & 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 & 0 \\ 0 & 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 & 0 \\ 0 & 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 & 0 \\ 0 & 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 & 0 \\ 0 & 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -12 & -13 & -8 \\ 6 & 4 & -6 \\ 8 & 2 & -8 \end{bmatrix}$$

Notice the convolution operation with stride is still the same process as previously presented. Since we said $n_p = 1$, our size of the Input image will change from 8×8 to size 10×10. In the previous subsection, the last 2 columns of the Input image were not utilized when producing the linear transformation. With padding, we were able to use the entire original Input matrix in the operation to produce the linear transformation.

5.1.2 Pooling

Pooling is utilized to reduce the size of computation and make some features detected more robust. We start with a 3D matrix $A = \{a_{i,j,k}\}$ with padding n_p of the size of $n_h^A + 2n_p \times n_w^A + 2n_p \times n_c^A$ and a 3D filter matrix B be the ones matrix in the size of $n_h^B \times n_w^B \times n_c^B$ where $n_h^B \leq n_h^A, n_w^B \leq n_w^A, n_c^B = n_c^A$. Note that there are no parameters to learn. This is a fixed function that the ANN uses as a form of reduction. The **pooling** matrix $P = \{p_{s,t}\} = A * B$ of A and B with a stride n_s gives a matrix in the size of $\lfloor \frac{(n_h^A + 2n_p) - n_h^B}{n_s} + 1 \rfloor \times \lfloor \frac{(n_w^A + 2n_p) - n_w^B}{n_s} + 1 \rfloor$ where

$$p_{s,t} = max(a_{i,j,k} b_{i-s+1,j-t+1,k})$$

for $sn_h^B - 1 \leq i \leq (sn_h^B - 1) + n_h^B - 1$, $tn_w^B - 1 \leq j \leq (tn_w^B - 1) + n_w^B - 1$ and $k = n_c^A$.

As an example, the *max pooling* of an input gray-scale (2D) 8 × 8 image and a filter of size 2 and stride of 2 is shown below:

$$\begin{bmatrix} 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 6 & 9 & 7 \\ 9 & 8 & 7 & 6 \\ 7 & 6 & 9 & 7 \\ 9 & 8 & 7 & 6 \end{bmatrix}$$

Essentially with max pooling, we are taking the convolution operation between the Input matrix A and filter matrix B of ones with a stride n_s. Once we have the submatrix, we then take the maximum value of the submatrix as the value of the pooled matrix. Notice padding was not implemented. It is not common practice to utilize padding with any type pooling. The purpose of pooling is to speed up computation while detecting strong features of A and the reduction of the size of the activations.

Another type of pooling is *average pooling* which acts in the same manner as max pooling with the exception of taking the average of the submatricies instead of the maximum value as shown in an example below:

$$\begin{bmatrix} 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \\ 5 & 7 & 0 & 4 & 9 & 1 & 0 & 7 \\ 9 & 1 & 4 & 8 & 3 & 5 & 4 & 1 \\ 1 & 3 & 6 & 0 & 5 & 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 2.5 & 5.5 & 4 \\ 3.5 & 4.5 & 5 & 3.5 \\ 4 & 2.5 & 5.5 & 4 \\ 3.5 & 4.5 & 5 & 3.5 \end{bmatrix}$$

5.2 Architecture of Some Commonly Used CNNs

Alex Net. Alex Net starts with a 3D Image matrix $A^{[0]}$ of size $227 \times 227 \times 3$ and filter matrix $B^{[1]}$ of size $11 \times 11 \times 96$ with a stride $n_s = 4$. (Krizhevsky) The stride works by approximately decreasing the size $A^{[0]}$ by approximately a factor of four drastically shrinking the dimensions of $C^{[1]}$ to $55 \times 55 \times 96$. This is then followed by max pooling with a filter $B^{[1]}$ size $3 \times 3 \times 96$ with stride $n_s = 2$ giving $P^{[1]}$ of size $27 \times 27 \times 96$. We then utilize the **same convolution** which keeps the original stride and utilize appropriate padding to obtain the same size between $C^{[l]}$ and $P^{[l-1]}$ with 256 filters of size 5. We obtain $C^{[2]}$ of size $27 \times 27 \times 256$. The architecture of the Alex Net CNN is shown below.

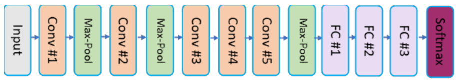


Figure 5.1: Architecture of Alex Net.

We utilize max pooling of filter size 3 with stride of 2 obtaining $P^{[2]}$ of size $13 \times 13 \times 256$. Using same convolution with 384 filters of size 3 obtaining $C^{[3]}$ of size $13 \times 13 \times 384$. We repeat the convolution prior to obtain $C^{[4]}$ of size $13 \times 13 \times 384$. We repeat same convolution with 256 filters of size 3 obtaining $C^{[5]}$ of size $13 \times 13 \times 256$. We utilize max pooling of filter size 3 with stride of 2 obtaining $P^{[3]}$ of size $6 \times 6 \times 256$ giving us a total of 9216 elements.

Unrolling the elements of $P^{[3]}$ into a column vector gives us a vector of size 9216×1 which is input into a fully connected (FC) layer containing 4096 nodes followed by another FC layer with 4096 nodes. We then feed that FC layer into the Softmax 1000 function obtaining out predicted output. This leads to approximately 60 millions parameters that needs to be trained. A model of the architecture of the Alex Net is seen below:

VGG-16. Instead of having as many hyperparameters as Alex Net, VGG-16 focuses on simplifying the network having convolution layers with the same filter size 3 with stride of one utilizing same padding and max pooling all with the same filter size 2 with same stride 2.(Simonyan) We start with a 3D Image matrix A of size $n_h^A \times n_w^A \times n_c^A$. The architecture of VGG-16 CNN is shown below.

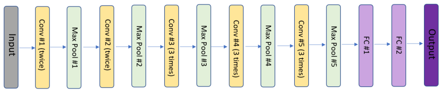


Figure 5.2: Architecture of VGG-16.

The first two layers of VGG-16 are same convolution layers with 64 filters obtaining $C^{[1]}$ and $C^{[2]}$ of same size. We follow this with a max pooling layer obtaining $P^{[1]}$ which is where the size reduction occurs. We follow the first max pooling with 2 same convolution layers with 128 filters obtaining $C^{[3]}$ and $C^{[4]}$. This is then followed by another layer of max pooling obtaining $P^{[2]}$. This is followed by 3 layers of same convolution with 256 filters obtaining $C^{[5]}$, $C^{[6]}$ and $C^{[7]}$ which is followed by another layer of max pooling obtaining $P^{[3]}$. We follow with 3 layers of same convolution with 512 filters obtaining $C^{[8]}$, $C^{[9]}$ and $C^{[10]}$ and max pooling obtaining $P^{[4]}$. This is followed with 3 layers of same convolution with 512 filters obtaining $C^{[11]}$, $C^{[12]}$ and $C^{[13]}$ and max pooling obtaining $P^{[5]}$.

Unrolling the elements of $P^{[5]}$ into a column vector is input into a fully connected (FC) layer containing 4096 nodes followed by another FC layer with 4096 nodes. We then feed that FC layer into the Softmax 1000 function obtaining out predicted output. This leads to approximately 138 millions parameters that needs to be trained. The downside is that there is a large number of parameters which needs to be trained, but is also very attractive because the dimensions decrease at a constant factor at the pooling layers only and similar convolutions are all trained together. A model of the architecture of the VGG-16 network is seen below:

Chapter 6

Experimental Results

6.1 Data Set

We are given two sets of 3D images pulled from the CIFAR-10 dataset: a training set X_{Train} that contains 7200 images and a testing set X_{Test} containing 1800 images. For X_{Train} 50% of the images are cat where the remainder are not and for X_{Test} 50% of the images are cat and the remainder are not. The dimensions of X_{Train} and X_{Test} are $7200 \times 32 \times 32 \times 3$ and $1800 \times 32 \times 32 \times 3$ respectively where the elements range from $[0, 255]$.

6.2 Data Preprocessing

We preprocessing the images in our data set by **standardizing** and **flattening** the images.

Standardizing. In data mining/machine learning, to reduce the bias created by different units/scales used on a quantitative data set, one usually wants to standardize entries in the input matrix into an interval between [0,1]. There are different ways of standardizing an input matrix. For the binary classification problem, we simply divide every entries in the input image matrices by 255.

Flattening. We also flatten each 3D image matrix in our data set into a 2D column vector done as the following: given a 3D RGB-image in the size of $n \times n$ where the three channels are given as

$$RGB(:,:,1) = \begin{bmatrix} R_{1,1} & R_{1,2} & \cdots & R_{1,n} \\ R_{2,1} & R_{2,2} & \cdots & R_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ R_{n,1} & R_{n,2} & \cdots & R_{n,n} \end{bmatrix}$$

$$RGB(:,:,2) = \begin{bmatrix} G_{1,1} & G_{1,2} & \cdots & G_{1,n} \\ G_{2,1} & G_{2,2} & \cdots & G_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ G_{n,1} & G_{n,2} & \cdots & G_{n,n} \end{bmatrix}$$

$$RGB(:,:,3) = \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n,1} & B_{n,2} & \cdots & B_{n,n} \end{bmatrix}$$

we form a flattened 2D column vector by collecting elements of each channel (in the order of R, G, B) first horizontally and then vertically:

$$x^{(i)} = [R_{1,1} \ \ldots \ R_{1,n} \ \ldots \ R_{n,1} \ \ldots \ R_{n,n} \ G_{1,1} \ \ldots \ G_{1,n} \ \ldots \ G_{n,1} \ \ldots \ G_{n,n} \ B_{1,1} \ \ldots \ B_{1,n} \\ \ldots \ B_{n,1} \ \ldots \ B_{n,n}]^T \in \mathbb{R}^{3n^2 \times 1} \quad (6.2.1)$$

For images in our data set, $n = 32$, thus by collecting all 7200 flattened training images together and all 1800 flattened test images together, we form 2D matrices: $X_{Train} \in \mathbb{R}^{3072\times7200}$ and $X_{Test} \in \mathbb{R}^{3072\times1800}$ for training data set and test data set respectively, as shown.

$$X_{Train} = \begin{bmatrix} \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(7200)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \end{bmatrix} \in \mathbb{R}^{3072\times 7200}$$

$$X_{Test} = \begin{bmatrix} \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(1800)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \end{bmatrix} \in \mathbb{R}^{3072\times 1800}$$

6.3 Experimental Results

We compare the performance of a simple ANN, AlexNet and VGG16. For the simple ANN, it has only one hidden layer with three neurons. For the AlexNet and VGG16, we apply parameters pre-trained from the ImageNet. We fine tune the hyperparameters and determine to use the following values of the hyperparameters for all three algorithms:

Total Number of Epochs	10
Initial Learning Rate	0.0001
Learning Rate Drop Factor	0.5
Learning Rate Drop Schedule	Piecewise every 3 Epochs
Mini Batch Size	60

The cross validation accuracy are summarized below:

Simple ANN	66.61%
AlexNet	81.83%
VGG16	83.28%

Our experiment shows that VGG16, the CNN with the most complex architecture, indeed obtains the highest classification accuracy. To further determine if the models we trained exhibits overfit problem, we plot the learning curves for the three algorithms given as

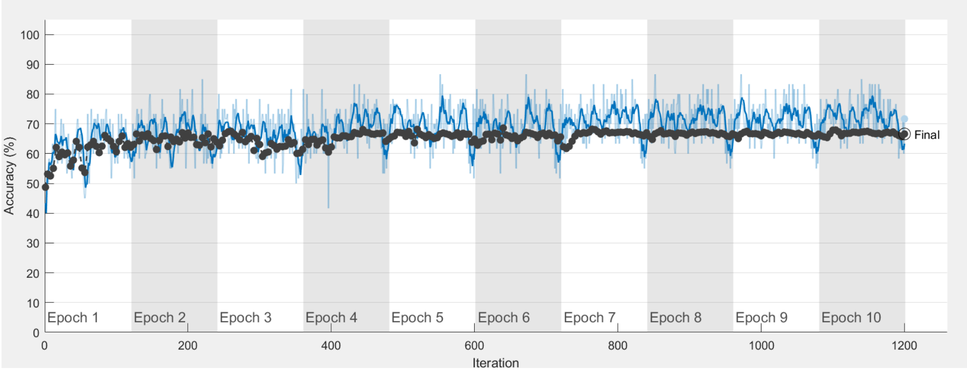


Figure 6.1: Learning curve of the Simple ANN.

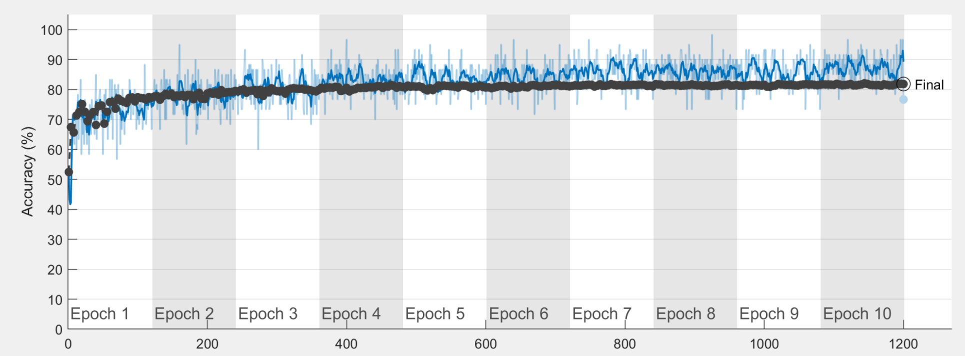


Figure 6.2: Learning curve of Alex Net.

The learning curves show that the models we trained does not exhibit the overfit problem: the accuracy on the training set and test set are close.

6.3.1 Results and Future Work

Looking at the above learning curves, we notice that in Epoch 1, the SANN has between 50 and 70 percentages where the Alex Net and VGG-16 learning curves start off pretty accurate, just under 65 to around 78 percentage. As we move through the iterations, we can see with the SANN the the accuracy never reaches above 70%. It seems to be on a steady increase with

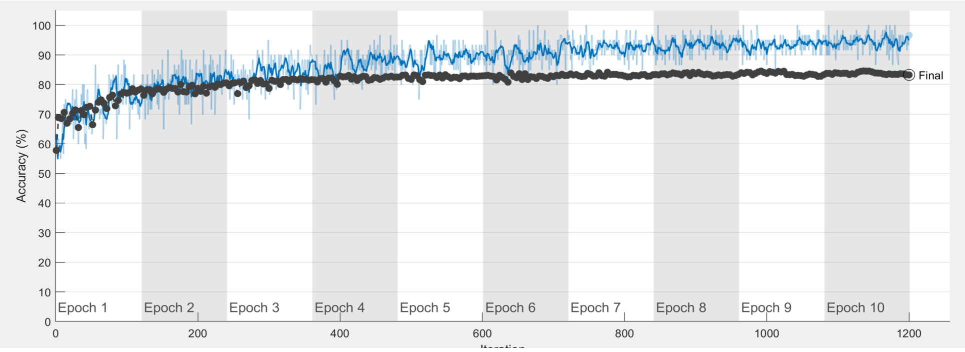


Figure 6.3: Learning curve of VGG-16.

accuracy during Epoch 1 then levels out reaching a final accuracy of 66.61%. With Alex Net, the learning curve increases during the first 2 Epochs then levels out reaching an accuracy of 81.83%. Utilizing the VGG-16 model, we see the learning curve increases at least during the first 3 Epochs then levels off to have a final accuracy rate of 83.28%. We can utilize CNN and other image analysis applications to stage various diseases, like Diabetic Retinopathy disease, that will help treat or prevent further complications. We could also place sounds on silent films and generating handwriting automatically.

Bibliography