Channel Islands

CALIFORNIA STATE UNIVERSITY

Performing Sentiment Analysis on Tweets: A Comparison of Machine Learning Algorithms Across Large Data Sets

A Thesis Presented to

The Faculty of the Computer Science Department

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Student Name:
Yite Lu

Advisor:
Dr. Brian Thoms

05 2019

APPROVED FOR MS IN COMPUTER SCIENCE

6-19-19

Advisor: Dr. Brian Thoms Date

6/25/2019

Dr. Michael Soltys Date

06-15-19

Dr. Jason Isaacs Date

APPROVED FOR THE UNIVERITY

6/26/19

Dr. Ozturgut, Osman Date

Non-Exclusive Distribution License

In order for California State University Channel Islands (CSUCI) to reproduce, translate and distribute your submission worldwide through the CSUCI Institutional Repository, your agreement to the following terms is necessary. The author(s) retain any copyright currently on the item as well as the ability to submit the item to publishers or other repositories.

By signing and submitting this license, you (the author(s) or copyright owner) grants to CSUCI the nonexclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video.

You agree that CSUCI may, without changing the content, translate the submission to any medium or format for the purpose of preservation.

You also agree that CSUCI may keep more than one copy of this submission for purposes of security, backup and preservation.

You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. You also represent and warrant that the submission contains no libelous or other unlawful matter and makes no improper invasion of the privacy of any other person.

If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant CSUCI the rights required by this license, and that such third party owned material is clearly identified and acknowledged within the text or content of the submission. You take full responsibility to obtain permission to use any material that is not your own. This permission must be granted to you before you sign this form.

IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN CSUCI, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT.

The CSUCI Institutional Repository will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Performing Sentiment Analysis on Tweets: A Comparison of Machine Learning Algorithms Across Large Data Sets

Title of Item

NLP, Machine Learning, Sentiment Analysis

3 to 5 keywords or phrases to describe the item

Yite Lu

Author(s) Name (Print)

Author(s) Signature

06/19/2019

Date

This is a permitted, modified version of the Non-exclusive Distribution License from MIT Libraries and the University of Kansas.

Performing Sentiment Analysis on Tweets: A Comparison of Machine Learning Algorithms Across Large Data Sets

Yite Lu

June 5, 2019

Abstract

Sentiment Analysis is a popular topic in machine learning, a subfield of computer science. In the past, Sentiment Analysis has been widely adopted in e-commerce and helps organizations analyze customer satisfaction of products and services. More recently, Sentiment Analysis has expanded its applications across government agencies as well, whether it being to analyze potential human threats within social media or political influence in election campaigns. Even more generally, humans are simply curious about how other humans are feeling. Two major approaches to Sentiment Analysis include lexical semantic analysis and machine learning. In this thesis, I will combine different word embedding techniques and use machine learning to analyze sentiments across published tweets. The overall goal is to discover which approach to Sentiment Analysis offers better performance.

Contents

List of Figures

1 Introduction

1.1 Natural Language Processing(NLP)

Since language was invented, it has brought significant changes to our daily life. In its early stages, people communicated with each other via speaking, and after the invention of "text", people started to communicate through writing. Texts have become one of the major media that people use to interact with others[60]. Text carries information including descriptions of things including who, what, where, when, why, and how. Scientists continue to explore the meaning and opinion of sentences[13][46] because it not only influences our behaviors but also has effects on decision-making[46]. As one example, a negative product review may affect customers' willingness to purchase a product, so understanding demands of customers continues to be an important part in business.[46].

The evolution of language includes the evolution of word formation, basic grammatical rules, and its lexicon[57]. More so, the size of a lexicon has a dependent relationship on the development of the civilization[29]. In the beginning, a civilization may start out with only a few hundred words, but as it develops, the number of words greatly expands. However, after reaching thousands of words, the scale of words increases very slightly because, for one, people have a harder time memorizing all the words and selecting appropriate words[61][60]. Consequently, words began to be categorized, leading to word ambiguity[8] where the same words can be expressed or understood as having more than one meaning[9]. Consequently, such words cannot be understood without specific context. For example, the word 'catch'.

- I catch a fish.
- There is a catch.

In order to eliminate word ambiguity, people require context dependent grammar in a sentence[8]. However, even with a context dependent grammar, problems can still exist.

Natural language processing(NLP) is the technique that uses computers to analyze and process human language[12]. Studying NLP, researchers are focused on analyzing syntax because it is the primary component of a

sentence structure[12]. However, semantic problems of NLP have inspired scientists to concentrate on using semantics rather than syntaxes[12]. This idea was affected by areas in the field of linguistic including grammar rules, parts-of-speech, and morphology[12][9]. Taking a simple sentence 'I like to eat apples' for example, computer scientists can use NLP to analyze this sentence and understand the meaning of this sentence and translate it into another language.

While traditional approaches help solve NLP problems within many applications, new challenges to NLP continue to arise. For example, retrieving latent information from user-generated content(UCG) or detecting deceptive phenomena such as fake news, is a challenging problem for NLP processors[12]. Moreover, language and grammar can have many exceptions, making it difficult for a computer to handle all exceptions correctly using the traditional NLP approaches.

The introduction of statistics into NLP processing has provided significant breakthroughs. In 1970, computer scientists discovered that there was an efficient way to use statistical methods[12], rather than only using grammatical and parts-of-speech analysis methods to process natural language[44]. With the development of computers and the tremendous growth of data, statistical approaches now offer new solutions to challenges in NLP.

These breakthroughs have made NLP a popular topic in recent years and has traversed many domains[12], such as sentiment analysis, machine translation, voice recognition, document classification, natural language generation, etc. Each topic has been broadly researched and many types of algorithms and applications have been developed. For example, to give a document and predict what kind of category does it belong to, that is the 'document classification'[85]. Giving a language model and the machine will generate new sentences[15] or a paragraph, even a new article(it has the ability to generate fake news), that is 'natural language generation'. 'Machine translation' is to translate one document into another language[58](for example, Google webpage translation). 'Voice recognition' accepting voice as an input and generate output texts[82] which can be used in creating movie automatic captions or a remote control device such as Amazon Alexa. Sentiment analysis has become an important application of NLP[46] because it is less complicated than other applications such as machine translation. The

topic is also interesting because people are generally interested in what others think and how they are feeling, particularly towards concrete objects such as commercial products, movies or books. For example, customers usually place an order online before they review the product review, understand how other customers' feeling about this product. Before, going to a movie theater, people sometimes search for comments of the movie on IMDB to decide whether they will watch the movie or not. Each of these has applications in sentiment analysis.

Sentiment analysis usually has a better performance on subjective text than on objective text[63] because subjective text contains more personal or emotional content[63][65]. It is the purpose of sentiment analysis to understand a users' sentiment-oriented contents. Objective text usually provides more neutral feeling, more depictions on things, and facts that do not consist of sentimental concepts. For example, 'touching a spider is scary', it is a subjective sentence showing a negative emotion. Conversely, 'California has the most population in the United States', is an objective sentence showing a fact that does not comprise any emotion.

Over the last decade, Twitter has become a widely popular social communication platform across the Internet[62]. Millions of tweets are generated every day, many of which consist of personal emotions[62]. Tweets that are posted by celebrities, enterprises or government officials have an unexpected influence on people. For example, if a celebrity tweets that she or he is fascinated in one specific brand of watch, people would have higher willingness to purchase this brand of watch rather than others. Even some malicious companies use tweets to manipulate people such as a mayoral election. Understanding the sentiment of tweets can help us to analyze what people are thinking and what aspects are people prone to. In business, companies can adjust strategies catering to customer's demands. In government, sentiment analysis can be used to evaluate the satisfaction of regulation or to win an election by understanding what voters are concerned about.

Researchers continue to work on sentiment analysis across numerous data domains. Pak and Paroubek[62] performed linguistic analysis approach to classify twitter data which contains objective and subjective texts. They used the presence of n-gram feature extraction. Pang and Lee[65] performed sentiment classification using machine learning approaches including Naive

Bayes, Maximum Entropy, and Support Vector Machines (SVM). They used the frequency and presence of unigrams and bigrams as features in machine learning approaches. Pang and Lee focused on a specific domain which was movie reviews and got the highest accuracy around 82.9% on SVM using the unigrams feature. Go et al.[20], they used the same machine learning approach as Pang and Lee. They scraped their own data thru the twitter API which the data was selected by emoticons. Then they used unigrams, bigrams, unigrams and bigrams, and parts of speech as features of tweets. Go et al. got the highest accuracy of sentiment analysis was around 83%.

In this thesis, I will perform sentiment analysis against public tweets for numerous reasons. Every tweet is a short sentence that has a limitation of 280 characters. In fact, only 12% of tweets are longer than 140 characters, so it is easy to analyze. As previously discussed, the complexity of analyzing the language is proportional to the volume of words. To construct a model for sentiment analysis, Python 3 was chosen, which has numerous third-party libraries that fully support a variety of applications. Python also provides an exceptional support in scientific calculation and data processing with visualized graphics.

More specifically, I used different types of artificial neural network approaches such as a simple artificial neural network (ANN), convolution neural network (CNN), and long short-term memory (LSTM). I tried bags of word vectors and word embedding vectors as features of tweets in machine learning approaches. The goal was to try different features across the amount of Twitter data and find the best performance machine. I did not collect my own dataset; I used the dataset from Go et al. instead. After all implementation, I got the highest performance of a machine was around 83.89% accuracy.

1.2 Environment setup

In this thesis, I select a machine learning approach to perform sentiment analysis. I performed an initial analysis using a MacBook Pro 13 and an integrated graphic card. To achieve better performance cloud computing was used. There are many cloud service providers in the market right now, such as Amazon AWS, Microsoft Azure, IBM Cloud, and Google Cloud Platform (GCP). I choose Google Cloud Platform as my cloud computing service. In

contrast to other cloud services, Google Cloud Platform provides 300 free credit balances for using cloud services, AWS and Azure mostly gives you 100 free credits. Additionally, Tensorflow is a free open source software that is used for machine learning and developed by Google, which are well integrated in GCP.

Before going through the machine learning, I setup two different VM instances in GCP. Both of them have a similar configuration except one has a GPU and the other does not. As far as we know, CPU and GPU have different design architectures. CPU is used to handle complex instructions and GPU has a better performance in the arithmetic calculation with a uniform type of data. Machine learning involves plenty of matrix operations making the configuration with GPU be a more optimal choice[59]. Figure 1 shows the different execution times of two machines while training with the same neural network. As depicted in Figure 1, using a GPU to train a model has better performance. Therefore, I can try different models and tweak parameters more efficiently to find a better model.

```
Train on 1149507 samples, validate on 127724 samples
Epoch 1/20
1149507/1149507 [==============================] - 1428s 1ms/step - loss: 0.4961 - acc: 0.7578 - val_loss: 0.4286 - v
al_acc: 0.7983
```

(a) training with CPU

```
Train on 1149507 samples, validate on 127724 samples
Epoch 1/20
1149507/1149507 [==============================] - 643s 560us/step - loss: 0.5021 - acc: 0.7540 - val_loss: 0.4415 -
val_acc: 0.7949
```

(b) training with GPU

Figure 1: Both machines train with the same model(A LSTM network with 100 neurons and a dropout layer,using 'sigmoid' activation function)

With the growing complexity of the model, the gap of execution time between training with GPU and CPU increases drastically. Table 1 shows the gap when the model becomes more sophisticated. This table only shows the time gap with one epoch. While we train a neural network, it usually takes 5-10 epochs or even more. Comparing to train a model which takes one day and one hour, the shorter one saves our time to try different approaches.

	LSTM(100 neurons)	LSTM(300 neurons)	3 layers CNN(300 neurons)
CPU	1,426s	1,805s	9,044s
GPU	639s	648s	900s

Table 1: The training time with the same model in different vm instance

In Figure2, I present a snapshot of the GCP dashboard.

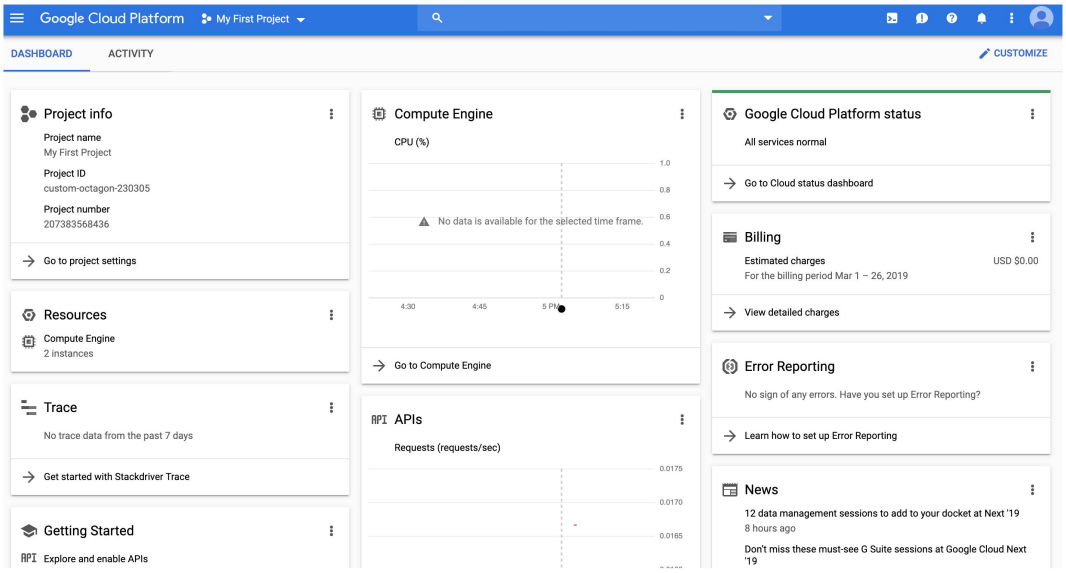


Figure 2: GCP dashboard

On the left-hand side of the dashboard, as shown in Figure 3, selecting the 'Compute Engine', we can create a Virtual Machine(VM) instance. In Figure 4, I can customize my machine settings or I can select Google pre-defined machines. GCP gives this flexibility to users. After settling down my configurations, I can review how much does it cost monthly on the right-hand side of Figure 4. GCP only charges you depending on resources while you are using.

Before I install the appropriate softwares, a couple additional steps are requried. First, because I created a VM instance with GPU, I have to install the driver CUDA (Compute Unified Device Architecture) for Nvidia GPU. Because GPU and CPU architecture are totally different, GPU is designed

to perform graphic operations,[45] if we want to execute our codes within the GPU instead of the CPU, we have to convert our codes into graphic tasks then submit into the GPU[18][6] It would be a difficult task and insufficient. Nvidia invents CUDA which is the library that convert codes into executable codes in the GPU to solve the packing problem. The version of CUDA must be compatible with my os system and version of Tensorflow. Second, Nvidia provides a package which is called cuDNN to accelerate calculation of training a neural network. Both drivers and packages can be downloaded from the Nvidia official website. Finally, I can step into the software installation part.

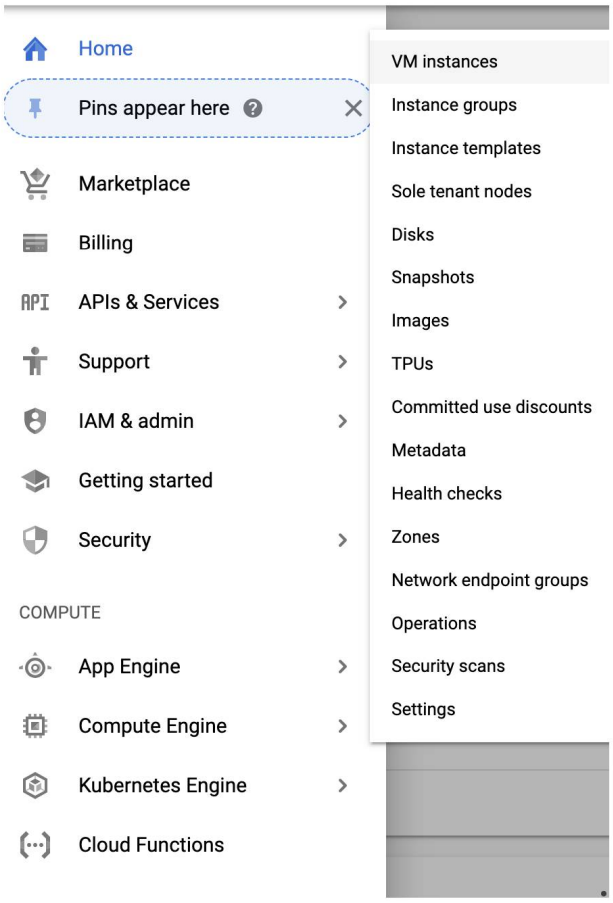


Figure 3: Cloud services list

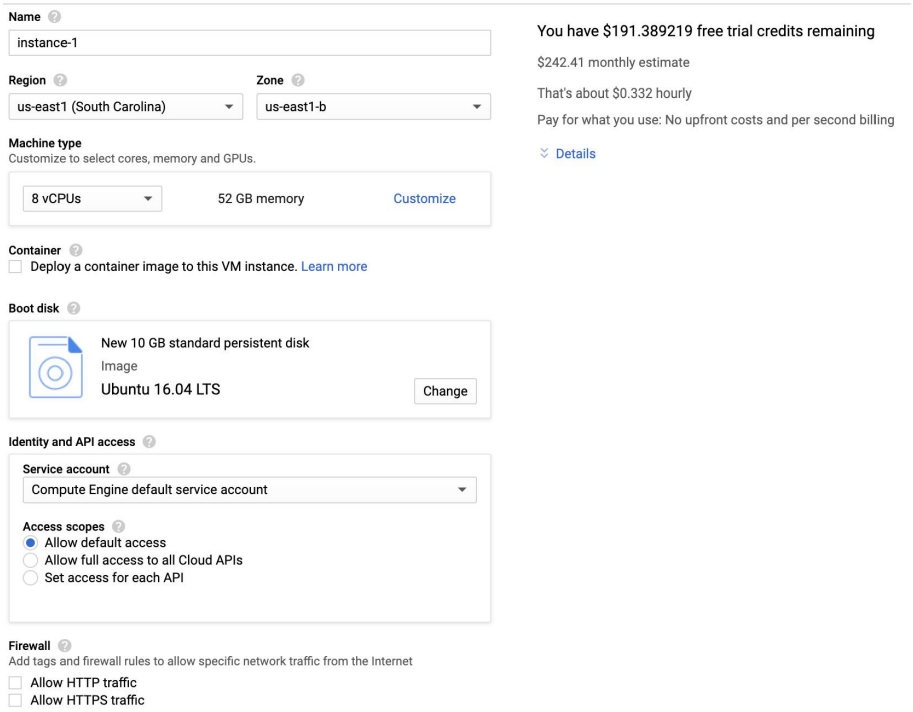


Figure 4: My VM configuration

Table 2 details the complete listing of all required software installed on the VM.

Numpy	Scientific computing,support N-dimension array data.
Pandas	Use the DataFrame structure to process data.
Matplotlib	Base on Numpy and pandas to support data visualization.
Sklearn	The basic machine learning framework.
Keras	A machine learning framework which is based on the Tensorflow.
Tensorflow	A popular open resource machine learning framework that released by Google.
Gensim	A package that is used to retrieve word vectors via unsupervised machine learning approach.
Tqdm	A small widget shows the progress while processing data in python.
NLTK	An entry level toolkit to process natural language.
Anaconda	A python package management sofrware.
Anaconda	A python package management software which helps programmers to create different virtual environments.
Jupyter Book	Providing a GUI that is freely interacting and visualizing result in python.

Table 2: Installed libraries

2 Background

In this chapter, I will introduce some terminologies that I used in this thesis. Before I use the machine learning approach to apply sentiment analysis on tweets, I have to pre-process the data. Because the machine learning approach cannot operate on raw data directly, I have to pre-process data and convert raw data into another type of data (usually numeric data such as an array type) as inputs then feed them into a machine. In section 3.1, I will introduce some terminologies about natural language pre-processing. In section 3.2, I will present the background and terminologies of the artificial neural network.

2.1 NLTK processing

The purpose of performing NLTK processing is trying to get feature of tweets and remove irrelevant data, also known as noisy data. After applying NLTK processing, I convert sentences that includes useful information such as sentiments into numeric type and input into a machine. Then a machine will find pattern in sentences to generate results.

2.1.1 Stop words

Studies show that removing stop words in sentences will increase the accuracy of sentiment analysis using the TF-IDF feature extraction[22]. In natural language, stop words refer to words that do not have useful or meaningful information and can be ignored without changing its whole meaning[9]. These words do not have positive impact on our algorithms, but they are considered as noises that might confuse the machine while training a model. Moreover, stop words such as 'the', 'an', 'a', etc, have high frequency that appears in a sentence or in an article. These meaningless words cost storage space and increase unnecessary computation. These words also tend to dilute word features in a sentence or an article. However, if we use the word vector representation such as Word2Vec as features, we do not remove stop words because certain stop words are related to the sentiment (negating words)[48]. Additionally, while training a word vector model, the relative position of words are important including stop words due to their training algorithm.

2.1.2 Parts-of-speech tagging

Parts-of-speech (POS) tagging is known as lexical categorization[9]. Using parts-of-speech tags as features in sentiment analysis has proven to be practical[36]. Words can appear in similar sentence with similar function. Also, words have morphological features. One word might have more than one parts-of-speech, such as a noun or a verb. For example, in 'I catch a fish' and 'There is a catch' apply the POS tagging, the results show [(I/PRP), (catch/VBP), (a/DT), (fish/JJ)] and [(There/EX), (is/VBZ), (a/DT), (catch/NN)] respectively (PRP=pronoun, VBP/VBZ=verb, DT=determiner, JJ=adjective, EX=existential there, NN=noun). We can figure out that word 'catch' was tagged to different parts-of-speech. A study showed the parts-of-speech combining with polarity of words are good features in sentiment classification[2]. However, in this thesis, I do not use parts-of-speech as features, but I will perform word lemmatization to retrieve the base or dictionary form of a word, thus, I have to perform parts-of-speech tagging first[9].

2.1.3 Stemming

In processing of natural language, we usually need to normalize the text and truncate any affixes[9]. Preprocessing with word stemming and lemmatization have been found helping increase the accuracy of sentiment classification[33][42]. Word stemming is a process that reduces a word into its word stem or root. It might not be able to represent an existing word. In English, words have different morphologies. For example, while we want to figure out the similarity between words, 'apples' and 'apple', both of them should be considered as the same word. Applying a word stemming will reduce 'apples' and 'apple' into 'appl', and 'effective' will become 'effect'. Stemming follows their algorithm stripping affixes and it is faster than lemmatization. There are three major algorithms which are Porter Stemming, Lovins stemmer, and Lancaster Stemming. Porter Stemming is widely used and becomes a standard today[83]. However, in this paper, I performed a word lemmatization rather than a word stemming because it gave me a better classified accuracy.

2.1.4 Lemmatization

Lemmatization is another process that is used in natural language to discover the similarity of words. Different from word stemming, lemmatization will transform a word into another 'existing' word.[9][5]

For example, 'driving' and 'drove' will both transform into 'drive'. However, lemmatization is more intricate than stemming. Before applying lemmatization, we have to perform part-of-speech first, then lemmatization transforms a word depending on not only its affixes, but also its parts-of-speech. If a word is tagged with different part-of-speech, it may lead to different results. Both stemming and lemmatization are used to normalize words[9][4][38][47], but have different levels of complexity and involve different methods and implementations.[47]. Stemming is more focused on informational index domain[38][4], and lemmatization is widely used in data mining and context analysis[4].

2.1.5 Word embedding

In mathematics, the word 'embedding' refers to a mapping function F:$x \to y$. The function F conserves two properties, which are injective (each x maps into an if and only if y,vice versa) and structure-preserving (given two group x1 and x2, if $x1<x2$, then $y1<y2$). Word embedding is used to map words into an N-dimensions vector space. There are two advantages. First, in a vector space, we use cosine similarity[43][30] to determine resemblance between words[48]. In the Euclidean space, the cosine similarity is result of calculating the dot-product of two vectors. θ is the angle between two vectors. If the angle is smaller then the $cosine(\theta)$ is smaller which means these two vector are closer.

$$A \cdot B = ||A||\,||B|| cos(\theta)$$

$$similarity = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{i} A_i \times B_i}{\sqrt{\sum_{i=1}^{i} (A_i)^2} \times \sqrt{\sum_{i=1}^{i} (B_i)^2}}$$

In the formula above, A_i and B_i are components of vector A and vector B respectively. The value of similarity is from -1 to 1. If the value is greater,

the cosine angle is smaller which means two vectors are closer and more similar. This attribute applies in any dimensional vector space, especially in high dimensions[74], to help us determine how close are words, sentences, and even documents. Secondly, using a vector representation improves the performance while training a neural network. More similar words mean more inputs that increase the chances of training neurons.

The main idea of word embedding is to reduce their dimensionality (i.e. transferring words with high dimensions to a specific vector, or low dimensions). There are two substantive methodologies. One is 'one-hot representation' (or it is called one-hot encoding) and the other is distributed representation.

2.1.6 One hot representation

One hot representation(encoding) is a straight forward strategy to transform words into vectors. It converts the original variable into a multidimensional variable, classified by the original features that replace and quantize with new features value of 0 and 1. For example, assume we have three words, 'money', 'cash', 'cards', we will label each word as a three dimension vector. 'Money' for [1,0,0], 'cash' for [0,1,0], and 'cards' for [0,0,1]. In NLP, one hot encoding is usually based on the Bag-of-words(BOW) concept. BOW assumes every word that is independent in both a sentence or in a paragraph. For example, there are two sentences below.

(1) David likes to play basketball and Mary likes to swim.

(2) David also likes to watch movies.

We add up appearances of each word in sentences and result in key-value pairs (word:counts). {David:2, likes:3, to:3, play:1, basketball:1, and:1, Mary:1, swim:1, also:1, watch:1, movies:1}. There are a total of 11 distinct words in the list. Then we tag the value as 1 if the word is in the position in the sentence, and tag the value as 0 if the word is not. For example,

David	[1,0,0,0,0,0,0,0,0,0,0]
likes	[0,1,0,0,0,0,0,0,0,0,0]
to	[0,0,1,0,0,0,0,0,0,0,0]
⋮	

After we generate all vectors of each word, then we turn sentence(1) and (2) into sentence vector by summing up each word vector. Sentence(1) and sentence(2) are shown as [1,2,2,1,1,1,1,1,0,0,0] and [1,1,1,0,0,0,0,0,1,1,1] respectively. As previously discussed, each word in BOW is independent, if there are two words 'like' and 'likes', they will present as two different vectors. Even though one hot representation is obvious and easy to implement, there exists two disadvantages.

- One hot encoding does not consider relative positions of words in the sentence.
- The size of word vectors might be very tremendous and waste plenty of space. (because it is a sparse matrix with plenty 0 and one 1)

The first downside considers the following sentences.

(1) Boys love girls.

(2) Girls love boys.

The word 'Boys' is denoted as [1,0,0], 'love' as [0,1,0], and 'girls' as[0,0,1] and capitalization is not considered. Sentence vectors are both denoted as [1,1,1]. While, in on sense, they are they same, in another they can be seen as totally different. In fact, in NLP, not only the position of words are important, but also each word is not independent (remembering BOW treats each word as independent). Thus, we might lose some features in the sentence.

A second problem considers the number of dimensionalities. If the number of words is large, the number of dimensions is going to be substantially larger. In machine learning, the number of features increases results in a proportional increasing dimensionality. Intuitively, we think more features will help the machine to create a better performance classifier. However,

in fact, when we are training a machine, we do not have unlimited training data, data sets in the vector space will be very sparse. The distance between two objects is likely very far away. While the number of features exceeds a specific value, the performance will begin to decrease[80]. That is referred to as 'the curse of dimensionality'.

2.1.7 TF-IDF

TF-IDF[66][3][86] stands for term frequency-inverse document frequency which is a weight factor technique that is used to evaluate the importance of a word in a corpus[66]. It is another feature extraction technique of texts in natural language[3]. It contains two parts, the first part is 'term frequency' and the second part is 'inverse document frequency'. The value of TF-IDF is to multiply them together.

$$TF = \frac{how\ many\ appearances\ of\ a\ word\ in\ a\ document}{total\ word\ counts\ in\ a\ document}$$

$$IDF = \log(\frac{total\ number\ of\ documents\ in\ the\ corpus}{how\ many\ documents\ appear\ with\ the\ word})$$

$$TF - IDF = TF * IDF$$

For example, assuming we have 100 keywords in one document(A) and 100 documents in the corpus, the word 'love' appears 10 times in document(A), then the TF is $10/100 = 0.1$. If the word 'love' appears in 25 different documents in the corpus, then IDF is $\log(100/25) = 1.386$ (here, the base of log is mathematical constant e), TF-IDF=0.1386. In fact, the base of log of above equation can be vary, we can choose the base to two, ten or mathematical constant e. The idea to use log in IDF is to dampen its value. For example, if the corpus size is huge (one million documents) and only one document contains the target word in this corpus, the value of IDF will be large and it will affect the result of TF-IDF. Assuming we have a document, those kinds of stop words such as 'the', 'I', 'and', etc usually have higher frequencies in a document, but they do not have too much useful information or meaning while we are performing sentiment analysis. Using term frequency we filter those words that do not have useful information, and then we can use TF-IDF considering the rest of the meaningful words.

2.1.8 Distributed representation

Generally speaking, if a representation is not one hot representation, then we call it a distributed representation. Compared to one hot representation, distributed representation maps data into a lower dimension space. It was first introduced by Geoffery E. Hinton in 1984[24]. In natural language processing, the idea is trying to find a way to map each word into a shorter and fixed vector, and all word vectors construct the word vector space. In this space, two words are free to capture the similarity by calculating cosine similarity. If two words are closer than other words, this means the two words have similar latent semantics. For instance, consider four words 'woman', 'man', 'boy', and 'girl'.

woman	[1,0,0,0]
man	[0,1,0,0]
boy	[0,0,1,0]
girl	[0,0,0,1]

Each word is represented as an independent vector. Contrast to one hot encoding, we manually categorize four words into two features: age and gender. Table 3 shows the result. Thus, 'girl' is encoded as [0,0] and 'man' is encoded

	0	1
gender	female	male
age	child	adult

Table 3: Result of words that have be categorized

as [1,1]. Applying this mapping, the dimensionality is reduced from 4 to 2. Each vector is considered as a single input to the model. Now, the number of input is changed from 4 to 2 because originally we use 4 distinct vectors to represent four words ,and now we only use two vectors. While training the model, the word 'girl' helps training not only the word 'woman' but also the word 'boy' because 'girl' belongs to 'female' and 'child' feature respectively. Using the one hot encoding, 'girl' only helps to train itself because words are independent in one hot encoding.

2.2 Machine Learning in neural network

Machine learning is widely used in a variety of domains including sentiment analysis[64]. A well-designed machine can perform tasks involving with computers. Using machine learning approach handling sentiment analysis problem is more efficient than using lexical sentiment analysis[64]. Machine learning uses training data as inputs and attempts to learn from the data as the human could. Essentially, a machine is a mathematical function (equation). The goal is to find an appropriate fitting equation that covers the most of the data. For example, feeding an image of a cat, the machine can determine that the image, with a certain level of certainty, that the image is a cat. This type of machine learning is known as image recognition; Another type of machine learning involves voice recognition. Given a voice recording file, the machine can distinguish what is this voice talking about. A machine defeats the top player in a 'Go' game after training, which is 'Alpha-Go'[14].

There are countless applications have been applied in real-life. Traditional algorithm applications receive data as input and result in outputs depending on the logical design of the algorithm. Nevertheless, machine learning focuses on the part 'learning by itself'. Once feeding the training data, the machine will train itself without human intervention. By feeding more data as inputs, the machine corrects and tweaks itself by feedbacks of output result continuously in order to improve its performance. This approach depends not only the algorithm but also accumulated data. Different from the traditional algorithm, with new coming data, modify the program is necessary.

2.2.1 Artificial neural network

There are many machine learning techniques that have been created, such as support vector machine(SVM), decision tree, Naive Bayes classifier, random decision forests, and etc. Artificial neural network(ANN) becomes the most popular and significant machine learning recent year because it is well-researched and developed, and attains extraordinary achievements in different domains[88]. For example, the CNN (Convolutional Neural Network) achieves dominance of image recognition[37][73][1] [19][23] and RNN (Recurrent Neural Network) has a outstanding success in speech recognition[52][19]. Compare to the conventional applications, even there exists a good application work well in a specific domain, it cannot be used in outside its

domain[32][88]. Moreover, the ANN has some benefits that do not exist in classical algorithms, such as adaptive, fault tolerance, massive parallelism, contextual information, learning ability and etc[23][32][88][79].

The ANN was firstly announced in 1943 by the neurophysiologist Warren McCulloch and mathematician Walter Pitts and aim to replicate the human brain[50]. In biology, the neural system is composed of neurons and each neuron receives electrical impulse via an axon and pass signals to each other. To determine whether a neuron will pass its own signal or not depends on how many signals does the neuron receive. If a neuron receives a sufficient number of signals(threshold), then the neuron will fire its own signal. To simulate this behavior, scientists design the neuron using an activation function. Receiving inputs and passing them into an activation function, to make a decision to pass a signal to next neurons. Next layer of neurons become new inputs of another layer of neurons, and so on.

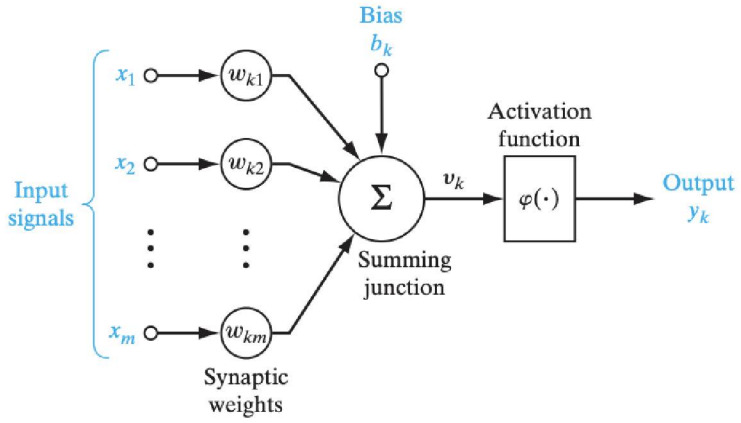


Figure 5: The structure of the perceptron[23]. x_1 to x_m are input signals, w_{k1} to w_{km} are the weights of the perceptron, and b_k is the bias. Output value y_k can be written as y_k=$x_1w_{k1} + x_2w_{k2} + \cdots + x_mw_{km} + b_k$

The perceptron is one of the simplest structures of ANN (figure 5). In this figure, $x_1, x_2, ...x_n$ are input signals, $w_1, w_2, ...w_k$ are the weights of each neuron, b_k is a fixed value which is a bias. Input signals will multiply with each weight and sum up with bias then pass into the activation function(ϕ). If the value of the activation function exceeds the threshold, an output sig-

nal will be fired[50]. The perceptron model can be used to solve basic linear discriminant problems. However, a number of weakness of perceptron had been discovered that it could not solve a trivial problem such as XOR classification[71][19][23][32]. In the real world, most problems mirror a XOR function, which are not linear separable[19][23][32]. Therefore, research in ANN has been stuck for many years without any breakthrough progress[32]. After stalling, scientists discovered that stacking multilayer of perceptrons could solve the non-linear separable problems[71][23].

In 1986, Ruineihart, Hinton, and Williams proposed a groundbreaking research 'backpropagation' which massively reduced the computation of training a neural network with multi layers[71]. The ANN will be trained if and only while the error is generated. The error is the discrepancy between the value generating by the machine and the expected value. The ANN usually spends tremendous computation on calculating errors. Backpropagation is an efficient algorithm to compute error and tweak weights of neurons within the ANN[71][72]. Research in neural network became a hot topic again. However, researches in neural network faced a major problem which is 'gradient vanishing' while using backpropagation to train a neural network more than 3 layers that caused activation useless[23]. Gradient vanishing means the gradient of the ANN will disappear and neurons cannot propagate signals to other neurons leading the network to stop training[26].

Fortunately, due to factors, such as increasing computational power of computers, big data, many computer scientists are exploring new algorithms of ANNs and achieve breakthroughs.

2.2.2 Type of machine learning

Machine learning algorithms can be broken into four types: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning.

Supervised learning needs two types of data: training data and labeled data. Labeled data is the desired output of training data[23]. The machine uses labeled data as a teacher that compares to the output generating by the machine to adjust and retrain itself. It is under supervised by labeled data which the naming comes from. For example, logistic regression and neural

network use labeled data to predict or classify data are types of supervised learning.

In contrast to supervised learning, unsupervised learning does not need labeled data[23]. It usually uses clustering, dimensionality reduction, and association rules. For example, k-means clustering and principal component analysis categorize data into many groups without any labeled data.

In the real world, it is tedious work to getting labeled data. Semi-supervised learning uses a small amount of labeled data and a greater number of unlabeled data to train a machine. For example, Google Photos application use a small amount of personal photos in your album and it automatically categorize other photos with name tagging.[19]

This type of learning varies from previous learning algorithms. It uses an agent to maximize the total rewards while learning. The core concept is trial and error[76]. The agent will try different decision during the training to get rewards. In every decision will not only affect the current step but also influence the next decision[19]. For example, Alpha-Go uses the reinforcement learning.

2.2.3 Underfitting and Overfitting

Once the kind of neural network is decided, the data set must be prepared and appropriate machine learning algorithm assigned. Take the supervised learning for example, feeding inputs x and generate the output y, the excepted value is y_1, training the the machine continuously until the loss which is $z = y_1 - y$ as small as possible. In every training process (one epoch), the ANN will tweak the weight of each neuron in order to generate a new output y and reduce the loss. After the machine finish the training process, the next step is to evaluate the machine. The test set to estimate the performance of the machine is used, because in the training process, the machine will find patterns within input data and tend to make the machine fit with input data. However, the goal of machine learning is to find a generic model that can fit in non-specific data (no matter it is training or test data). The estimated result of the machine comprises three conditions: underfitting, overfitting, and appropriate fitting.

Underfitting and overfitting are two major problems while training a machine. Underfitting means data and model cannot match perfectly. A machine does not have enough features of data or training time is not enough[19].

The solution is trying to add more features, reducing the degree of normalization of data, and increasing training epochs.

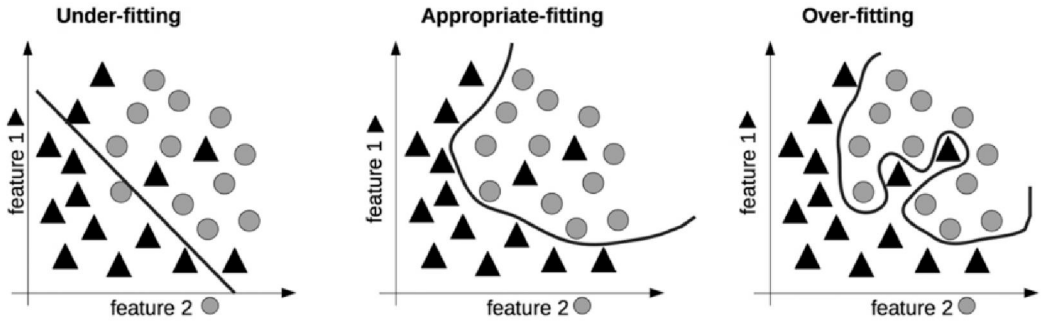


Figure 6: Different fitting situations. Finding a appropriate equation relies on experience.[17]

Overfitting is also a serious problem. It usually happens in a model that contains too many features which make the machine is too sensitive to data. It is overtraining on the training data that causes a worse performance on the real data (test data). The problem can appear in ANNs, especially in deep learning[39][75]. There are many methods to avoid this, such as adding a dropout layer in the neural network[75], getting more training data, increasing the degree of normalization[31][75], and early stopping[23]. As I mentioned in the curse of dimensionality, too many features lead to overfitting that's why performance is decreasing. While training the machine, we use cross-validation data in a training set to monitor whether the model is underfitting or overfitting[23].

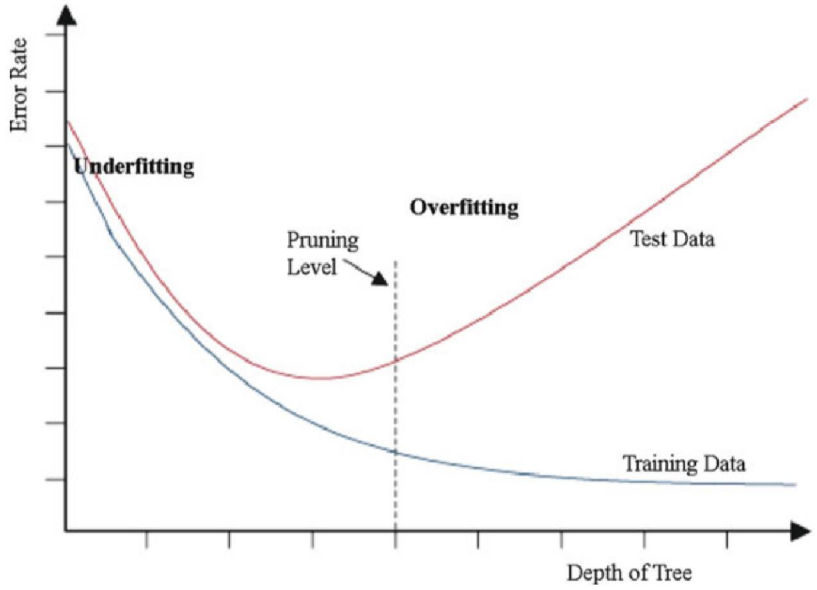


Figure 7: The value in y axis represents the error rate and the value in x axis represents the training epochs. If the error rate of test data begin to raise and error rate of training data is still going down, overfitting is supposed to happening.[16]

2.2.4 Cost function and Regularization

As mentioned previously, the goal of machine learning is to try to find an appropriate equation that is compatible with data. Therefore a "cost function" is used to optimize the machine and find the best model of the machine. The goal is to minimize the cost function. It means the error between the machine we generated and we desired that is the smallest. We can represent a machine in a generic format showing below[19].

$$y = a_0 + a_1 x^1 + a_2 x^2 + \cdots + a_n x^n$$

In machine learning, input data can be described in vector format, thus I rewrite the equation into the vectorized form $y = A \cdot X$.

- A is the model's parameter (weight) vector including the bias a_0 and feature weight from a_1 to a_n.
- X is the instance's feature vector from x_0(always equal to 1) to x_n.

The next step is to define a cost function. Taking a simple linear regression for example, I define a cost function using Mean Square Error (MSE). The example shows in figure 8. The green points are the desired values and red points are the value that generated by the model. The distances between each green point and red point are the error that we generated. These errors should be as small as possible. If another equation is found where the cost function is smaller, we can say the new machine is better.

$$cost\ function = MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y_i}\right)^2$$

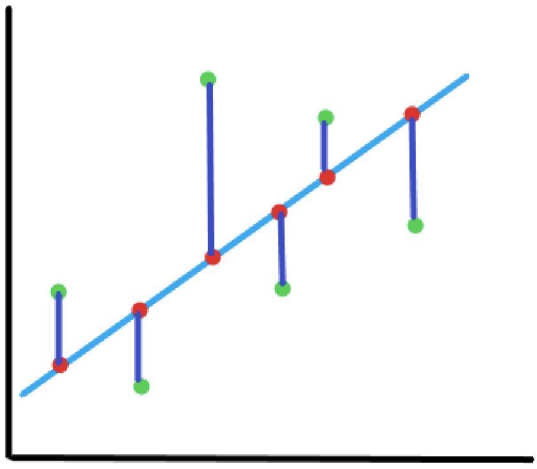


Figure 8: Cost function example. Green points are the expected values and red points are the actual values. Blue line is the machine that we found. The loss of each instance is the distance between green point and red point.

Figure 9 shows two machines (equations), assuming the black one is machine(1), the blue one is machine(2), and green dots are data. I want to find a machine (equation) that can classify the data.

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 \quad (1)$$

$$y = a_0 + a_1x + a_2x^2 \quad (2)$$

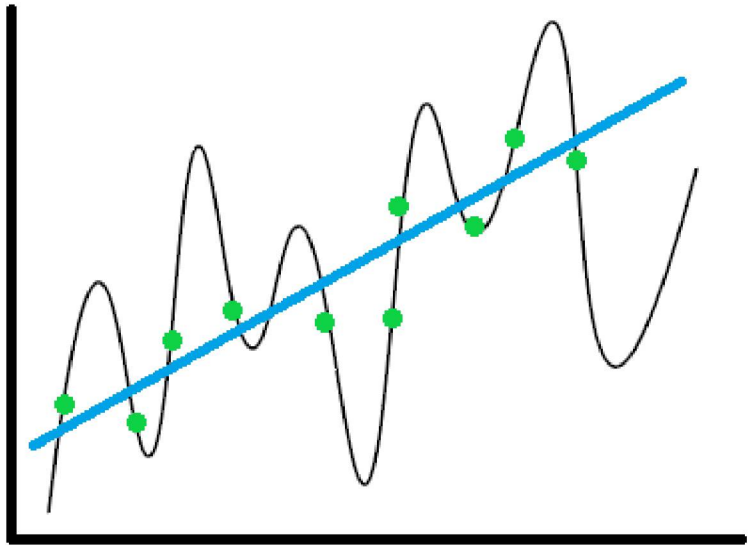


Figure 9: Even the black equation classifies the data (green points) perfectly, we want to find the blue line instead because the blue line is more generic to data. The black one is too sensitive to data and might get a worse performance on the new coming data.

The goal is to figure out the coefficients of each machine. To take a glance of these two equations. Even though equation (1) matches data perfectly, we prefer using a straight line which is equation (2) to classify the data. Apparently, equation (1) seems overfitting. In machine learning, finding a simple model rather than a sophisticated model is preferred because a simple model is more generic to data. Machine (1) has a good performance in classifying training data but it may result a worse performance than machine (2) with the new coming data.

The L1 and L2 regularization are two tricks to prevent the machine to be overfitting. They both add a penalty part to the cost function. I use the same cost function (MSE) for example showing below. L1 regularization adds the absolute value of coefficients to be the penalty and L2 regularization appends the square value of coefficients as the penalty (λ is a constant value).

$$L1 \quad cost\ function = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \sum_{i=1}^{n} a_i x^i\right) + \lambda \sum_{i=1}^{n} |a_i| \tag{3}$$

$$L2 \quad cost\ function = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \sum_{i=1}^{n} a_i x^i\right) + \lambda \sum_{i=1}^{n} a_i^2 \tag{4}$$

Taking equation (1) for instance. Calculating the cost function, we can see that the coefficient of a high order has a stronger impact on the cost function than a lower order coefficient. a_5 has a stronger impact to cost function than a_0 and a_1, even a_5 is a small number, it will dominate the output value of the equation and influence cost function. In order to minimize the cost function and assuming $\lambda = 1000$, I will minimize coefficients in high order, otherwise the cost function cannot be minimized. After coefficients in high order have been minimized, the equation reduces the degree and looks like the lower degree. Therefore, it solves the overfitting problem. Imagining if the $a_3x^3 + a_4x^4 + a_5x^5$ part of the equation (1) has been removed, it will look similar to equation (2) and has a similar shape. The only difference is the value of coefficients in a_0, a_1, a_2.

2.2.5 Gradient descent

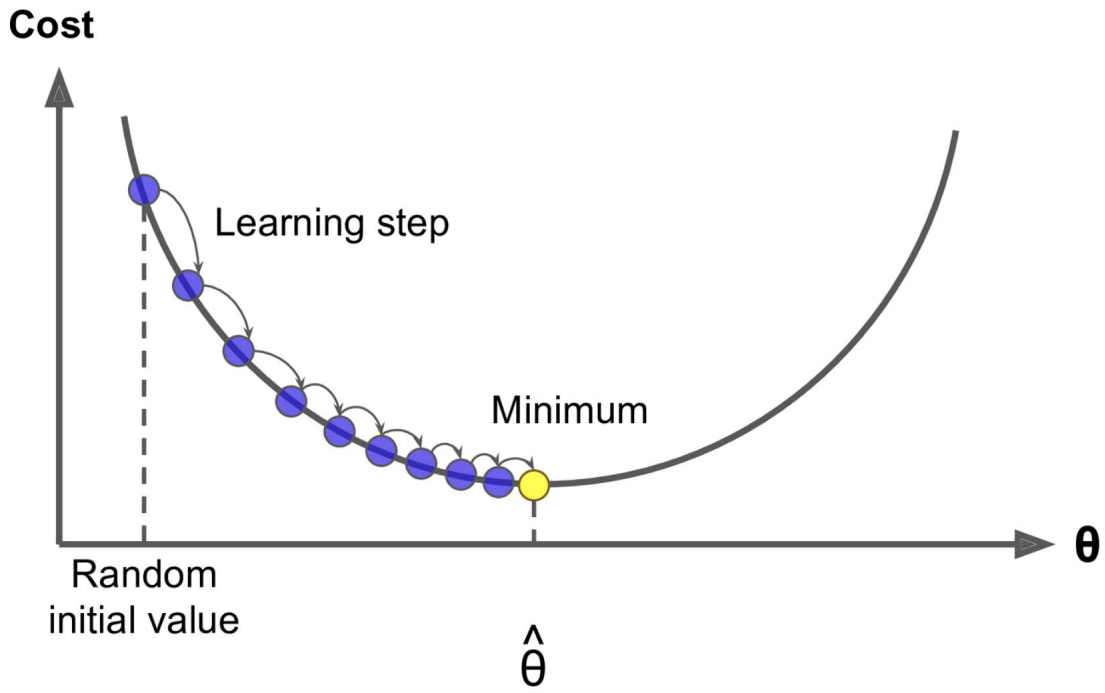


Figure 10: Gradient descent:starting with a random value and move toward the opposite direction of largest gradient[19]

Most machine learning algorithms need optimizations and gradient descent can be used to adjust the parameters of the model[23]. Because there are tremendous amount of parameters in the ANN, using the gradient descent algorithm help us short the training time and tweak parameters in an efficient way[21]. Optimization means changing the x to maximize or minimize the value of the function $f(x)$. This is done using a cost function. Gradient descent is the partial derivative and represents the steepness degree in every dimensionality of the point x in the space[23]. A gradient of multi-dimensions vector x is to calculate the partial derivative of each element of x. $\nabla f(x)$ is denoted as the gradient of a multidimensionality vector.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

The direction of the gradient is the same as the highest changing rate of a function f(x) which means in that direction, function f(x) increases or changes the most. Therefore, if we want to minimize the cost function efficiently, we must move toward the opposite direction of the gradient. While gradient becomes to zero, it means you get the minimum cost function which is-known as gradient descent algorithm. An easy understanding example asks "If you are in the mountain and you are blind, you want to find a path to reach the bottom, how will you do?"[19] Intuitively, you may move toward to the direction of the steepest slope step by step and finally, you will reach the bottom.

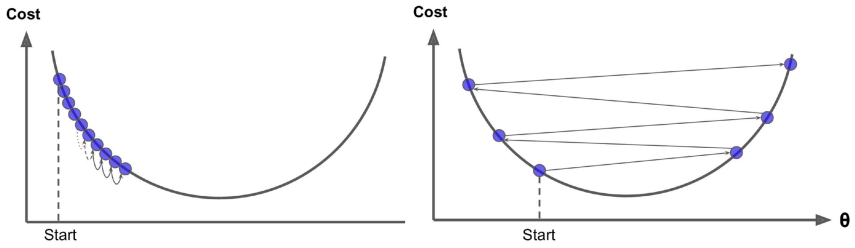
$$gradient \nabla = \frac{\partial f(\theta)}{\partial \theta}$$

Gradient descent algorithm shows as below

$$\theta = \theta_0 - \eta \nabla f(\theta_0)$$

θ_0 is the variable, θ is the update value of variableθ_0,∇ is the gradient, $f(\theta)$ is the function, and η is the learning rate.

There are many types of gradient descent approaches, such as batch gradient descent, stochastic gradient descent, and mini-batch gradient descent. It usually begins with a random initial value (or a random point in the space), and takes a baby step to calculate the gradient, then decides the next step by direction of the gradient. The purpose is to tweak parameters that makes cost function as small as possible. While training a machine, the learning rate is a hyperparameter that will affect the gradient.



(a) small learning rate (b) leanring rate is too big

Figure 11: Different learning rate[19]. When the learning is too small, the machine might not be able to reach the minimum. Conversely, if the learning rate is too big, the value of the gradient will bounce back and forth, and it cannot reach the minimum either.

The learning rate in (a) is too small, even the model has trained for many epochs, gradient still does not reach the minimum. Conversely, learning rate in (b) is too big, that causes the gradient bouncing back and forth and cannot reach the bottom.

However, there is a pitfall of gradient descent. It has already been proved that using gradient descent in a convex function will finally reach a local minimum which is as same as the global minimum of the convex function.[70] Nevertheless, in a non-convex function the worst case, gradient descent does not guarantee let you will get a global minimum, instead, you only get a local minimum (it might have a chance to be the global minimum too), the gradient converges at a saddle point[41].

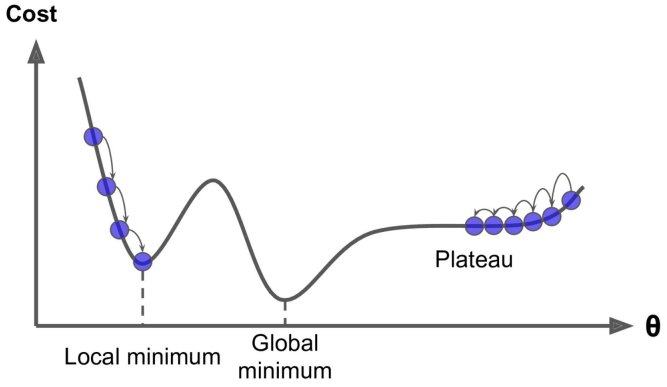


Figure 12: non-convex function might converge at a local minimum, not the global minimum[19]

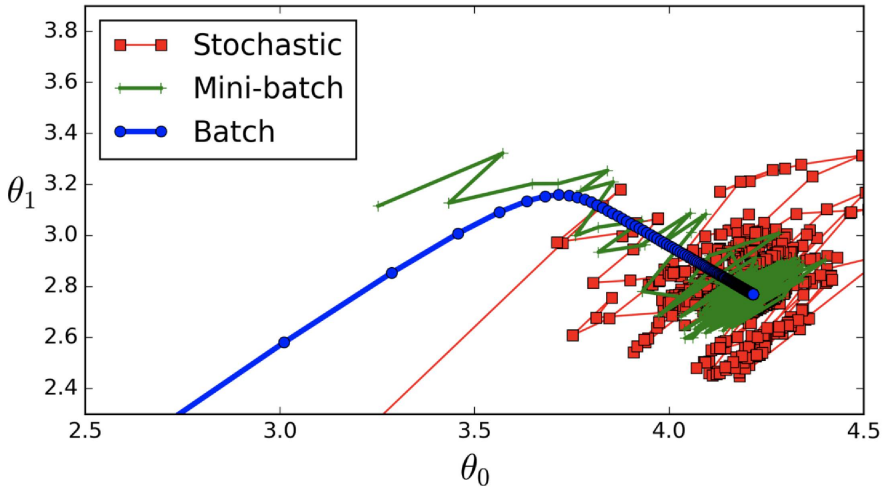


Figure 13: different learning path to a minimum[19]

2.2.6 Backpropagation

Backpropagation is the algorithm that usually combines with the optimization algorithm such as gradient descent to train the neural network[23][72]. It use the chain rule for derivatives in calculus to accelerate the speed getting the gradient[72]. The first created ANN is perceptron. It doesn't have any hidden layer, only inputs and outputs layer, which cannot solve the non-

linear discriminant problem[23][72]. A few years later, scientists notices if we add one more hidden layer to the network, it is possible to handle these type of problems[55][72][23].

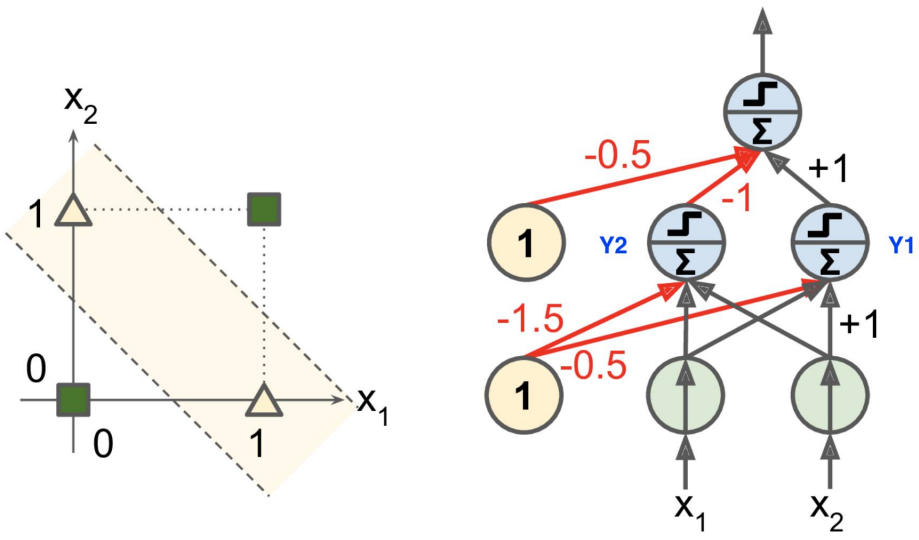


Figure 14: XOR problem (a non-linear separable problem) can be solved by using the MLP[19](with one more hidden layer than perceptron).

$$Y_1 = X_1 + X_2 - 0.5 \tag{5}$$
$$Y_2 = X_1 + X_2 - 1.5 \tag{6}$$
$$output = Y_1 - Y2 - 0.5 \tag{7}$$
$$activation \quad f(v) = \begin{cases} 0 & if\ f(v) < 0 \\ 1 & if\ f(v) \geq 0 \end{cases} \tag{8}$$

Multi-layer perceptron(MLP), it contains the hidden layer and uses the feedforward propagation to train the machine. Every neuron in the network is an activation function and each neuron is connected together as a network. Feeding data into inputs and pass thru each neuron to outputs.(figure15)

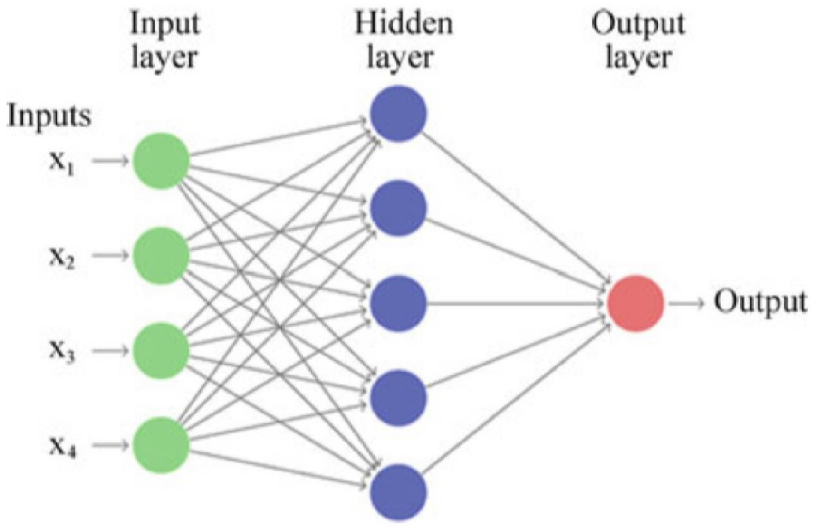


Figure 15: Feedforward network structure[16]

Before the backpropagation algorithm was announced, calculating the gradient of the loss was the bottleneck of training the ANN[71][72][23]. With the growing number of hidden layers and neurons, finding the weight of the neurons becomes challenging. Because we have to perform a bunch of computation of the gradient of the loss, then depending on the loss to tweak the weight. Using the backpropagation algorithm reduce the computation of calculating the gradient and decrease the time of training the ANN. We first propagate signals forward to each layer of neurons and generate the loss. The loss is the variance between the output value generate by the machine and the expected value. Next step is to calculate the gradient of the loss and backpropagate error signals to previous layer, and then tweak the weights of neurons[71][72].

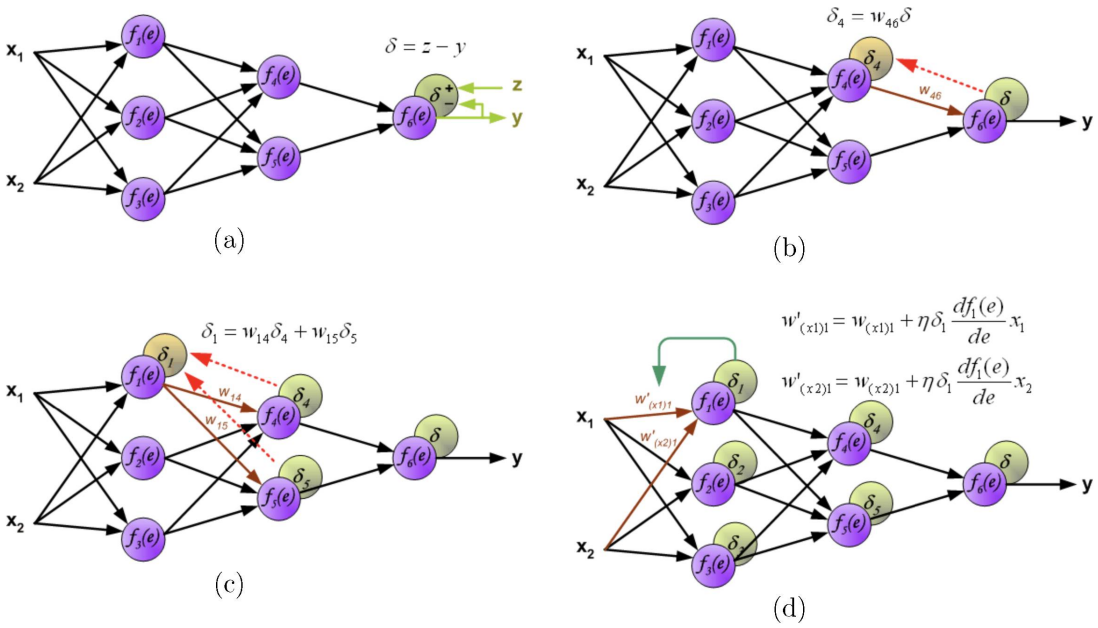


Figure 16: Backpropagation progress:(a) Calculating the error signal δ = desired value(z) - output value(y) (b) The error signal δ backpropagate to previous layer. Because δ comes from neuron$f_4(e)$ and $f_5(e)$, the error δ will backpropagate proportionally by the weights of w_{46} and w_{56}. Once we get the δ_4, then propagate the new error δ_4 to previous layer again (c) If a neuron receives more than one error signal, then sum up them (d) After reach the input, then tweak weights of every neurons[7]

2.2.7 Activation function

The activation function is an essential part of a ANN. It determines whether an input signal will propagate to other neuron or not. It ensures a neural network to be intricate, thus, the network can learn more features or structure complex data, work like a human brain. Choosing an appropriate activation function is important. In a neural network, if an activation is linear, then output will be also linear because of every calculation just different linear combination. Therefore, we need a non-linear activation function instead. A sigmoid function was the most widely used one[23][72][49]. Its shape resem-

bles an 'S' which is why it is called 'Sigmoid'.

$$y = \frac{1}{1 + e^{-x}}$$

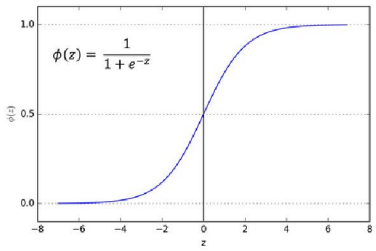


Figure 17: Sigmoid activation function: The output value of the sigmoid function is between 0 and 1.

Sigmoid has some basic properties. (1) Its domain is all real numbers. (2) Its output is between 0 and 1, so it can compress a large number into a small number. (3) It is a bounded, differentiable function. However, while the input is either a very large or small number, the output value of the sigmoid function change very slow. Its derivative is close to zero which mean the gradient is close to zero. While applying the backpropagation algorithm, the derivative of current layer relies on previous layer's derivatives. That leads to a situation in backpropagation, output value is always zero and neurons will not be trained.[23][26][27]. It is called 'saturation', especially it happens in a deeper neural network. Another problem is that sigmoid is non zero-centered which means the output value is between 0 and 1. However, the range of actual data in the input layer may lie between a and b. After applying a sigmoid function, the properties of distribution is no longer preserved.

While sigmoid is deprecated and the neural network is deeper and deeper, more and more activation has been created. Now, the most popular activation is ReLU and Tanh. Especially, ReLU function is widely used in deep learning[23][1][56][49]. It has some advantages that compare to sigmoid. First, it is easy to compute so it is faster than sigmoid. Second, it does not exist the gradient vanishing problem while propagating forward signals[84]. Nevertheless, it does have a disadvantage. In the training process, if a very big gradient passes through the ReLU function, which leads to

update the weight significantly, the ReLU have a chance of dying. Because if the updated value is smaller than 0, when the signal pass through another neuron will always be 0. These neurons die forever and will not be activated again.[56][84]. Fortunately, researchers have discovered update versions of ReLU and ReLU is still widely used nowadays[84].

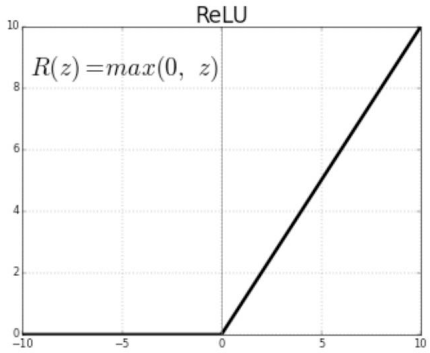


Figure 18: ReLU: The activation outputs 0 when $R(z) < 0$ and outputs z when $R(z) \geq 0$

2.2.8 Word2Vec

In the previous section, I have mentioned distributed representation. In this section I will introduce the Word2Vec which is a distributed representation model training by the neural network. Word2Vec converts words to vectors in a distributed representation way. It was firstly announced in 2013 by Mikolov, Chen, Corrado, and Jeff Dean[51] while working at Google as reseachers. Even before this research was published, many researchers have tried to create a model converting words into vectors. However, most models were not very efficient, they were either time consuming or complex architecture[51][54].

Word2Vec uses two architectures, Continuous Bag-of-Words(CBOW)[51] and Skip-Gram[51] to build the solution. It contributes lots of breakthrough research in natural language processing in recent years because it uses an unsupervised training to generate distributed word vector, not to generate a predicted model of a neural network. Placing these word vectors in the vector space, if we want to find another similar word, we just need to calculate the cosine similarity and figure out which word is closer. It is not very intuitive

if we just directly look at two words. Moreover, another paper is published by Mikolov, Le, and Sutskever that shows even in different language, similar words which are scattered in a vector space, their relative positions in the space are similar[53]. It proves that using a distance to describe the similarity between words in a vector space is reasonable.

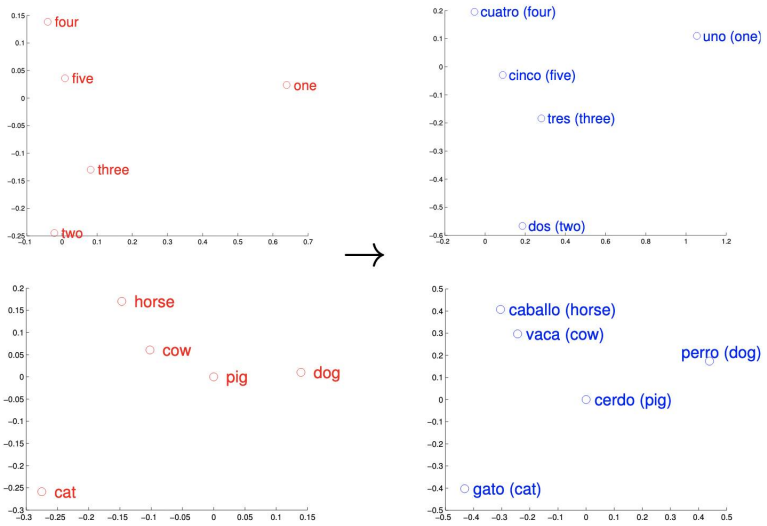


Figure 19: English (on the left) and Spanish(one the right) shows two type (numbers and animals) of word vectors that scatter in vector space and their relative positions. (word vectors have been projected down in two dimensions and rotated.[53]

In contrast to other models, Word2Vec contains two advantages, training is fast (because it uses an unsupervised training) and architectures are simple (CBOW and Skip-Gram both are 3 layer network).

2.2.9 CBOW

In Word2Vec, giving a context as inputs and trying to predict the word itself as an output is called the CBOW model. This model comprises three layers, one input layer, one projection(hidden) layer, and one output layer. In figure 20, we assume there are five words and w is the target word, using two previous words w(T-2), w(T-1) of w and two following words w(T+1), w(T+2) of w to predict word w. Inputs use one hot encoding and they share

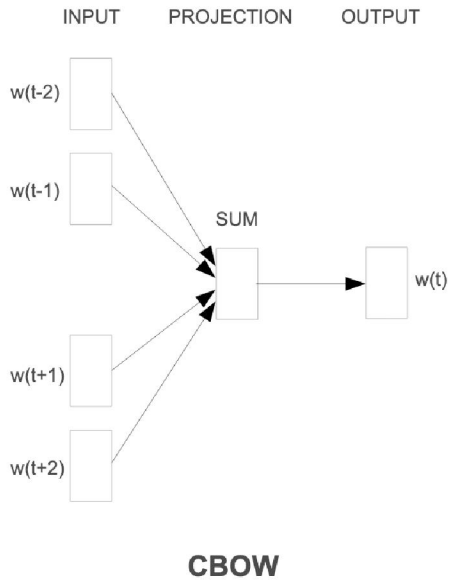


Figure 20: CBOW model[51] use nearby words of the target word to predict the target word. In this figure, window size equals to 2 which means how many words nearby the target word. w(t) is the target word, and w(t-2),w(t-1),w(t+1), and w(t+2) are words nearby the target word.

the same weights in a projection layer. Here, we say predict target word w, the probability of w is not the point but is used to adjust weights in the projection layer. Briefly speaking, the transit matrix between projection and output is the weight matrix that is equivalent to the word vector.(W' matrix in Figure 21)

This model uses backpropagation and stochastic gradient descent(SGD) tuning weights[51]. For instance, let's consider the following sentence. 'Today is very hot'. The corpus size is 4 (because we only have 4 words). We want the output word vector in 3 dimensions. Then, 'Today' is encoding as [1,0,0,0]. After training the network, we get a weight matrix whose size is 4×3. Output suppose to be 1×3. Usually, the neuron's number is equal to the dimensionality of word vectors and the number of dimensionality is far less than the number of vocabulary size. Simply speaking, this is an operation reducing the dimensionality from one hot encoding to distributed

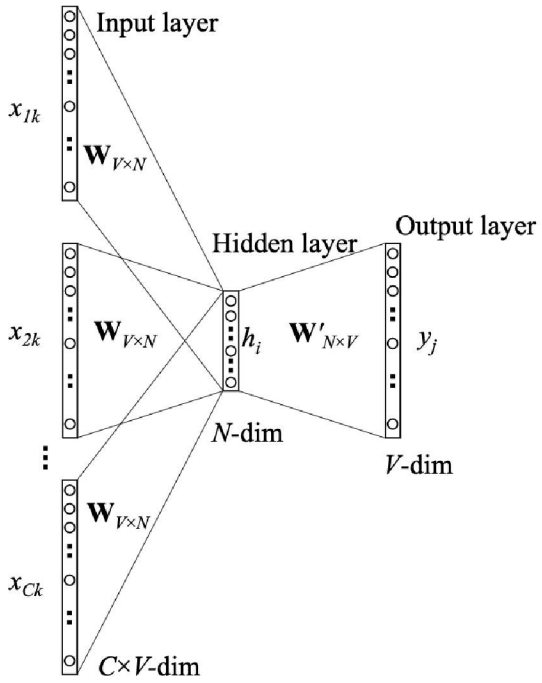


Figure 21: Weight matrix $W'_{N\times V}$[69] is the word embedding matrix. Input vector $x_{1k}, x_{2k}, \cdots, x_{Ck}$ are nearby word vectors that using one hot encoding and y_j is the target word vector using one hot encoding.

representation.

word 'Today' | Weights after training | Word vector of 'Today'

$$[1 \quad 0 \quad 0 \quad 0] \times \begin{bmatrix} 1.45 & 3.21 & 0.87 \\ 2.22 & 1.65 & 3.87 \\ 1 & 2.08 & -1.04 \\ -2.6 & 4.55 & 7.12 \end{bmatrix} = [1.45 \quad 3.21 \quad 0.87]$$

Figure 22: CBOW example. Assuming we get a word matrix, the word embedding vector of the word 'Today' will be the multiplication of two matrices.

2.2.10 Skip-Gram

Another architecture in Word2Vec is Skip-Gram model shown in Figure 23.

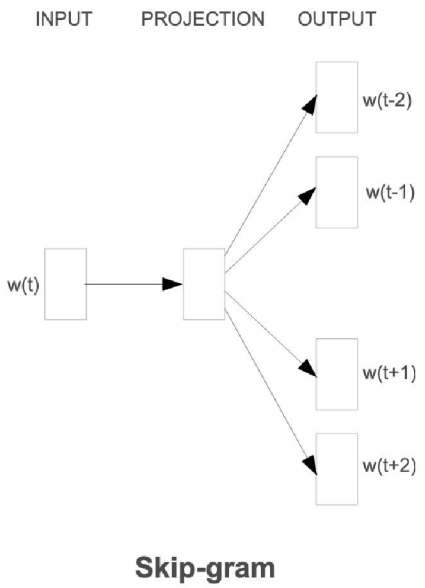


Figure 23: Skip-Gram model[51] use the target word to generate nearby words. w(t) is the target word and w(t-2),w(t-1),w(t+1) and w(t+2) are nearby words of the target word.

Different from CBOW, Skip-Gram takes the reverse approach, using a word as an input then predict the context of the word. The architecture is similar to CBOW and contains three layers, input, projection, and output but runs in opposite fashion. For example, let us consider the following sentence, 'Machine leaning is a black box technology' and assuming context windows = 2. The size of context window determines how many words nearby the target word will be used in predicting.

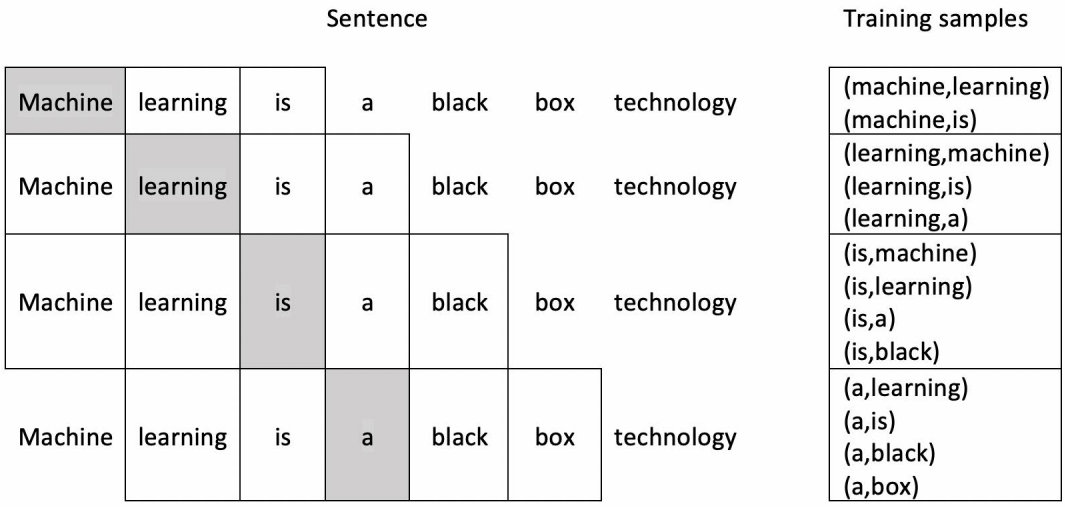


Figure 24: Skip-Gram example. The window size equals to 2 and each target word generates pairs of target and nearby word pair.

The neural network uses target word producing training samples and using them to perform the same training approach (backpropagation and SGD in CBOW) in figure 24.

After adjusting the weight matrix(between input layer and projection layer), we will get word vectors (the weight matrix W in Figure 25).

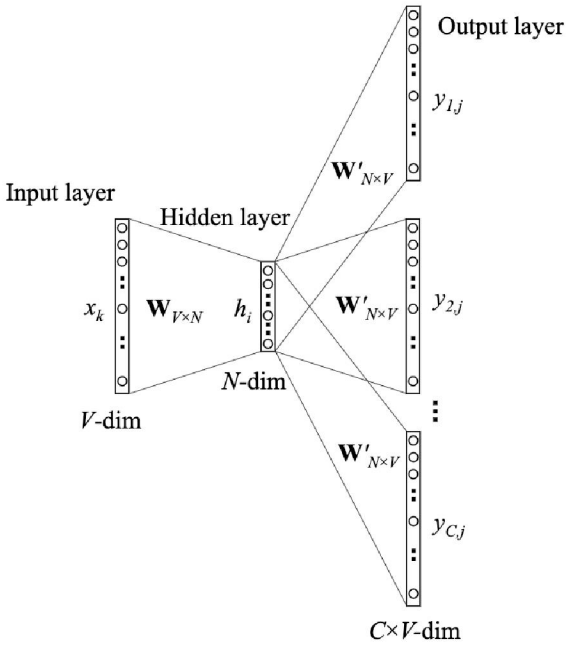


Figure 25: Weight matrix $W_{V\times N}$(not W')[69] is the word matrix. x_k is the target word using one hot encoding and $y_{1j}, y_{2j}, \cdots, y_{Cj}$ are nearby words of the target word using one hot encoding.

To conclude, CBOW model uses nearby words to predict the probability of center word and then tune the weight continuously to get the whole word vectors after finishing training. Tuning the weight is universal, so the gradient affects all nearby word vectors. The prediction number is almost the same as vocabulary size. Skip-Gram model uses the center word to predict nearby words, and adjust the weight continuously to get the result. It is clear that Skip-Gram model takes more time than CBOW model because every center word has to perform the prediction every k times (if the window size is k). Therefore, the training time of Skip-Gram is much longer than CBOW.

However, in the Skip-Gram model, every word is affected by nearby words. While training the model, every word becomes a center word that will perform k times adjustment and prediction. If the vocabulary size is smaller or some words rarely appear in a corpus, Skip-Gram usually generate a more accuracy word vectors because it has been tuned and predicted more times.

Even word in CBOW model is also influenced by nearby words, nevertheless, the gradient is averagely distributed on nearby words. Infrequently occurring word did not get a special fully training.

2.3 Convolutional Neural Networks

Convolutional neural networks(CNN) is a significant neural network model especially in deep learning in recent years. Usually, the most suitable application we use with CNN is image recognition. Surprisingly, studies show that CNN is also adapt to NLP[34][37][87]. The model is created based on the human brain's visual cortexes and how they operated in image recognition.[68] There was a well-known competition called 'ImageNet' that was used to evaluate the performance of the ability to classify images of a machine. In 2012, AlexNet which is create by a team with members Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton that used combination of techniques, such as adding dropout layers, making the network deeper(deep learning), training with parallel GPU, using 'ReLU' instead 'sigmoid' and data augmentation in CNN that had significantly performance increasing in this competition[37]. Since this competition had been held, the result did not have a remarkable breakthrough, error rate usually was around between 25% to 30% until the appearance of AlexNet. The AlexNet received 16.42% error rate that defeated rank 2 with 26.22% error rate. Thus, AlexNet shows that CNN is capable to fit in a complex structure of a network, also becomes a milestone of deep learning.

Basically, a CNN constitutes three types of layer, convolutional layer, pooling layer, and fully connected layer and we can stack these layers as many as we want (of course we have to use some skills to prevent overfitting) to create a deep neural network. CNN involves a lot of computation while training the model. Because of these computational complexities, it was not very practical and seems not achievable. Fortunately, with increasing performance of hardware and optimizations of GPU for computation, CNN makes tremendous progress.

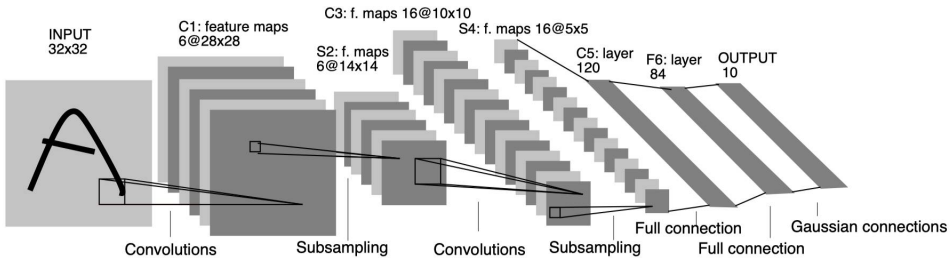


Figure 26: CNN structure example with convolutional layers, pooling layers, and full connected layers(LeNet-5)[40]

2.3.1 Convolutional layer

First of all, what is convolution? Generally speaking, convolution is a mathematical operation that use two functions f and g to generate a new third function.

$$h(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau$$

With the changing x, $h(x)$ becomes the new function that is the convolution of $f(x)$ and $g(x)$.

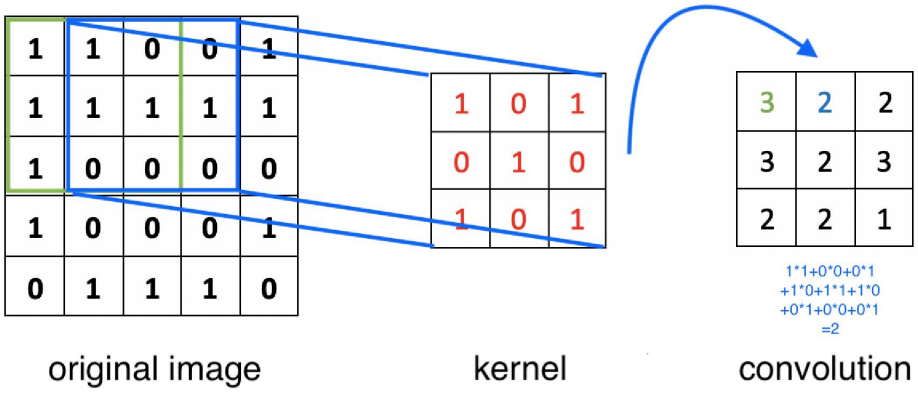


Figure 27: example of convolution operation

As we know, an image consists of pixels and each pixel is represented a numeric value. A human's eyes is a perfect recognizer and we can easily distinguish the difference between a dog and a cat in one image, the machine

does not have this ability. This is because our eyes perform feature extraction automatically when we see something thru our eyes. Therefore, convolution is used to perform feature extraction in a CNN. Therefore, we can use the CNN to perform feature extraction in NLP[34][37][87].

When we want to perform convolution in a picture, the original image is $f(x)$ and we need another function $g(x)$ which is we called 'kernel'. After applying the convolution of $f(x)$ and $g(x)$, we extract features in an image such as contour. Based on different kernel methods that generate different effects on the original image. The effects include sharping or smoothing the original image, finding out contours, and figuring out shapes. Then CNN uses distinguishable convolution operations that extract all kind of features and stack them together as a classifier at the end.

2.3.2 Pooling layer

The goal of pooling layer is to reduce the quantity of data and reserve the most meaningful feature information[10]. CNN usually combines with deep learning which means a CNN has many hidden layers. While performing the convolution to retrieve features, as I mentioned, different kernel methods produce different feature maps which depends on the number of kernel methods. Pooling layers (or is called sub-sampling layer) decrease the dimensionality of data to prevent overfitting and data computation. In practice, CNN uses max pooling which means choosing the max value of a window (for example 2x2 size) as an output value, and average pooling uses the average value of the window as an output.

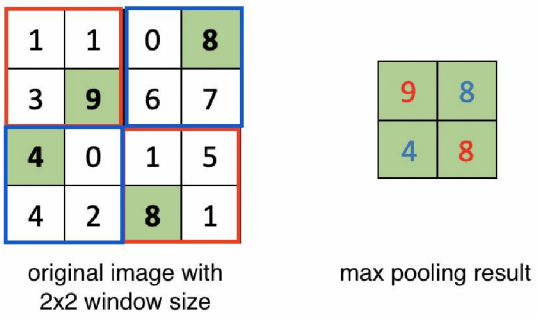


Figure 28: Example of max pooling operation. Maxpooling reserves the most important features.

2.3.3 Fully connected layer

Fully connected layer is used to be as a classifier in the CNN. Convolutional layer and pooling layer extract features from the sample data, mapping these features into a implicit vector space. The fully connected layer maps these extracting features into the sample data. It usually locates before the output layer. Figure 26 shows the mechanism

.

2.3.4 Dropout layer

Additionally, a dropout layer is a special layer that used in CNN, especially in deep learning to prevent overfitting.[75] It randomly discards a number of neurons, makes them disconnect to other neurons. This skill makes the network be more generic. While training the network, connected neurons do not depend on input from some specific neurons[25].

2.4 LSTM(Recurrent neural network)

Long short-term memory (LSTM) is a specific type of Recurrent neural network[28]. Some studies show this type of ANN improve semantic representation in NLP and sentiment analysis[77][81]. Since we already have an artificial and convolutional neural network, why we need a recurrent neural network? Because in ANN and CNN, we consider each object is independent, but in the real world, many types of data is dependent, it is sequential[19][23]. Therefore, RNN adds one more element which is the 'time series' and it can be used to predict the future. For example, the stock price of a company changes with time passing. A word in a sentence also has context relation. There are four types of RNN. (1) Input a sequence of data and get a sequence of output such as predict stock price. (2) Input a sequence of data and get a value as an output such as sentiment score. (3) Feed an input value and get a sequence as output such as giving a picture or a video, and generate caption as results. (4) The most complex one which uses an encoder (sequence to vector network) and decoder(vector to sequence network) such as translating one language into another language.[19]

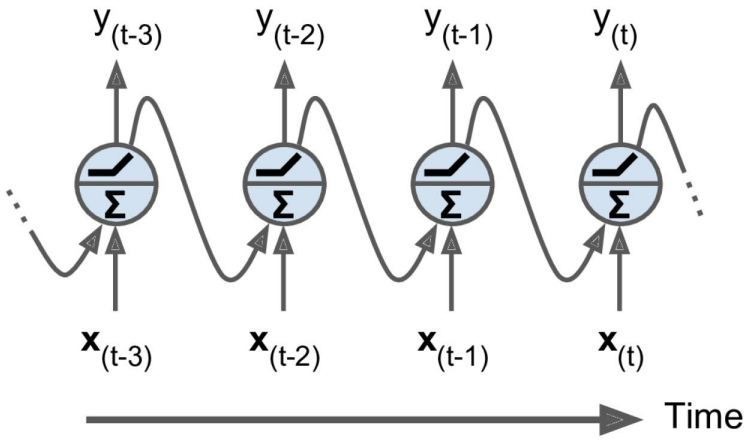


Figure 29: An unfold recurrent neurons example. On the right most neuron cell, it not only receives the input from x_t, but also receives y_{t-1} as inputs, t is the current time[19].

While RNN can be used in solving natural language because its property of time series, but it has a small drawback. In RNN, the closest time series cell has more influences on the current cell than a faraway cell. Thus, here comes out a LSTM which decides what kind of long and short term memory will be kept. It was firstly announced in 1997 by Sepp Hochreiter and Jürgen Schmidhuber[28].

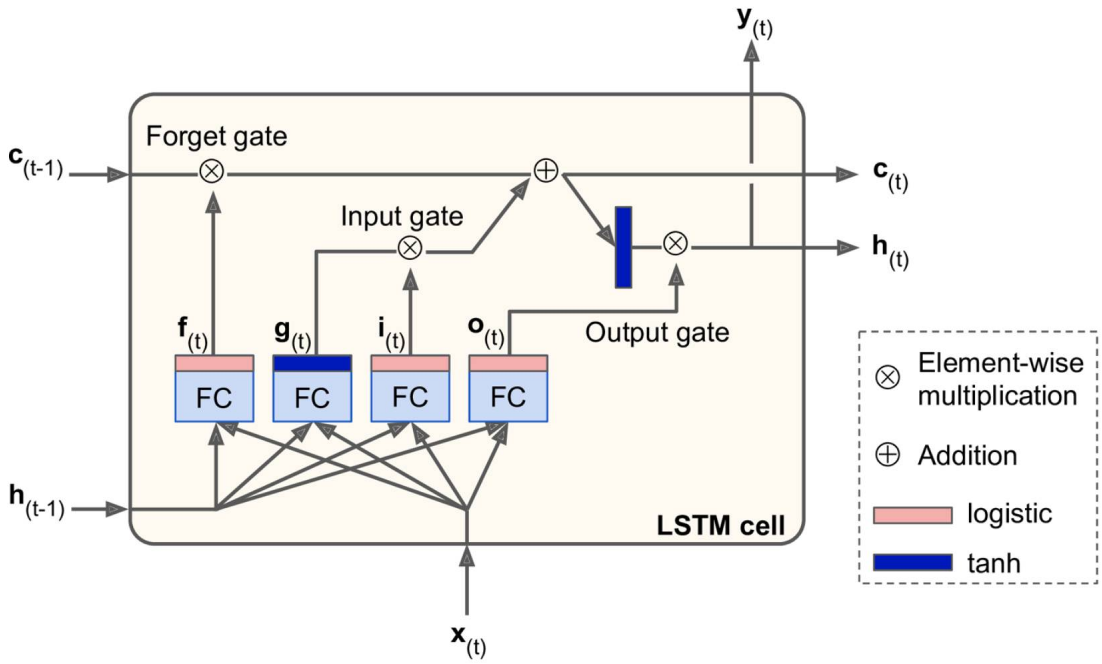


Figure 30: A LSTM cell[19].

In Figure 30, $c_{(t-1)}$ and $h_{(t-1)}$ are the long term memory and the short term memory of previous time stage $(t-1)$ respectively. In the current time stage (t), $x_{(t)}$ is the input and $y_{(t)}$ is the output. $c_{(t)}$ and $h_{(t)}$ are the long term and short term memory generating in the current time stage (t) and they will be used in next time stage (t+1). Forget gate decides which part of long term memory will be discard. Input gate decides which part of $g_{(t)}$ will be a part of long term memory. Output gate controls which part of long term memory will be the current output $y_{(t)}$ and current short term memory $h_{(t)}$. Because these three gate use the logistic activation function, the output is between 0 and 1. These gates use element-wise operation to decide whether the information can pass the gate or not. If the value is equals to 1, then the information can pass the gate, otherwise it cannot pass.

If we dig into the cell, it looks a little complicated. It uses three gates and extra input to determine how the memory will be stored and used. In Figure 30, c_{t-1} is represented as long-term memory, and h_{t-1} is the short-term memory.x_t is the current input. Forget gate accepts the input from the

previous cell and decides to keep or forget the long-term memory. Input gate decides the new memory which generated by combining current input and previous short-term memory that will add into a part of long-term memory. Output gate decides which part of long-term memory will combine with the current input as a new short-term memory[19]. To notice that each activation in a LSTM is irreplaceable[28].

2.5 Evaluate the model

The final step of machine learning is to evaluate the performance of a model. For this thesis, the performance is the ability to classify sentiment belonging to positive or negative. Accuracy, precision, recall, and F1 score are used to evaluate a classifier. For example, if we want to create number '5' classifier, for a positive example and its prediction is also positive, that is a 'true positive'(TP) and if its prediction is negative, that is a 'false negative'(FN). For a negative example, if its prediction is also negative, that is a 'true negative'(TN). Otherwise, it is a 'false positive'(FP)[19].

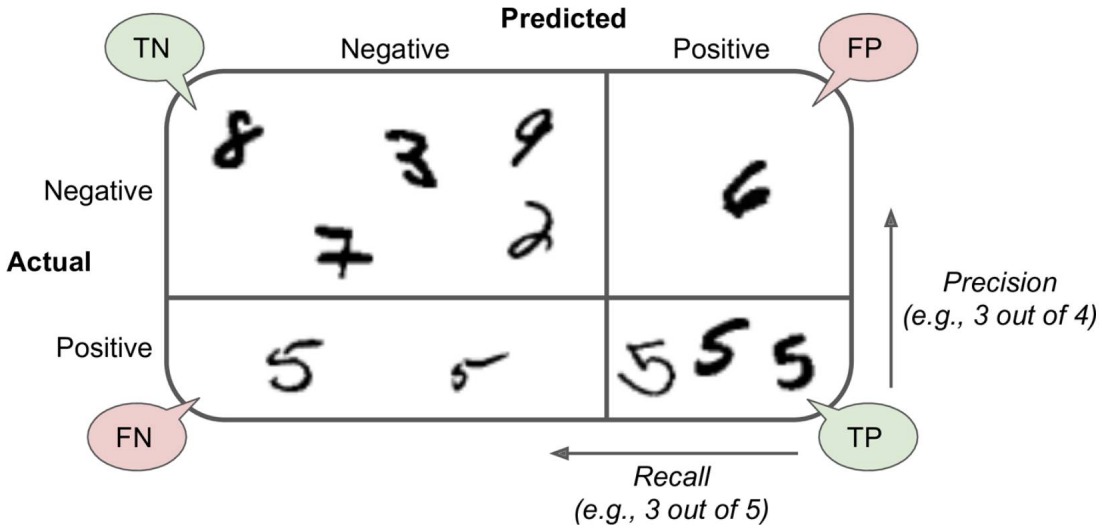


Figure 31: A 5 classifier, TP,TN,FN, and FP are showed above.[19]

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$F1\ score = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 * \frac{precision * recall}{precision + recall}$$

According to the formula above, precision is more sensitive to false positive(FP), which means precision try to get all positive examples can be recognized. Conversely, recall is more sensitive to false negative(FN), which means recall is used to evaluate how many positive examples have been ignored. Simply speaking, 'precision' is how many positive predictions correctly of all positive predictions, and 'recall' is how many predictions to be positive of all positive observations.

In different circumstances, we sometimes want precision higher and sometimes recall higher. For example, while predicting the earthquake happening, we want the recall is higher, which means we prefer every earthquake has been detected. Sacrificing the precision to detect all earthquake happening rather than missing a few earthquakes. Conversely, in an email spam system, we wish to have a higher precision and tolerate a low recall. A system recognizes a spam email as a normal is acceptable. Unfortunately, precision and recall is a trade-off[11]. In practical, we have to decide the decision boundary depending on current circumstances. F1 score can be used in exploring the unbalance data set. For example, assuming we have 99 negative and 1 positive data in data set, if the machine only output negative as a result, then accuracy is 99%. However, in this extreme situation, the performance of the model is useless. Therefore, we use f1 score as a harmonic mean of precision and recall which balance the value. In tradition, f1 score is between 0 and 1 and higher f1 score is better.

3 Implementation

In this thesis, I implement different machine learning approaches and word representation to find the best machine to classify sentiment on tweets. Moreover, I tweak each model to improve its performance. In this section, I will first use Word2Vec to create an embedding model then use a different type of ANN to implement sentiment analysis. While using machine learning, there is a basic flow chart. First, I have to pre-process the data. Next, I define a model and extract its features. Then I will define the structure of the ANN and train my algorithm. The next step will be to evaluate the machine with test data. Finally, I tweak parameters to get better performance or use a different structure of the machine.

3.1 Data Cleansing

Kaggle is a public platform which provides numerous type of data for machine learning. In this paper, I will use the data set 'Sentiment140 dataset with 1.6 million tweets' from Kaggle[20] to implement sentiment analysis on tweets. After downloading the data set, it has already been labeled. Let's take a glance on the data. It consists of a lot of, for the scope of sentiment analysis, unnecessary symbols such as '#' and '@' symbol. Also, words use latin-1 encoding. Making things more difficult, tweets are not a formal language, so it contains some meaningless words, emotional symbol, and words with repeating characters.

While performing clean data, I use a DataFrame structure from Pandas, which provides higher efficiency than using Microsoft Excel (limitation data row number in Excel is 1 million). After loading the data set into DataFrame, I covert all text into lower case and use regular expressions to remove unneeded symbols. Removing symbols and converting text to lower case will have a minimal effect on sentiment analysis.

The next step is to implement word stemming or lemmatization. Here, I choose using lemmatization which will generate intact words. Before applying lemmatization, performing the part-of-speech tagging is necessary. Some sentences in the data set consist only meaningless or weird words, after applying lemmatization, these sentences become empty. Therefore, DataFrame

was used to remove empty data row and save the clean data. The clean data consists few columns including 'sentiment', 'original sentence', 'lemma' which is the columns that contain a word list within the sentence, and 'clean_text' which joins all words in column 'lemma' to produce a new sentence. After data pre-processing, I am ready to create my own Word2Vec model by my own corpus.

	sentiment	text	clean_text
85	0	@imaginechurch Hi everybody! I Wish I was there Have a great day!!	hi everybody i wish i be there have a great day
86	0	I forgot to adjust the a/c before I went to bed so I just woke up all sweaty and hot. Gross	i forget to adjust the ac before i go to bed so i just wake up all sweaty and hot gross
87	0	is finishing her box of sees candies nuts & chews today! No more until I or someone I know goes back to the states.	be finish her box of see candy nut amp chew today no more until i or someone i know go back to the state
88	0	Looks like I'm gonna have to sell some stuff if I want that APC40. http://bit.ly/193hJ8	look like im gon na have to sell some stuff if i want that apc
89	0	@iTweetReply @tomricci I just got an email, it was just a whole lot of code	i just get an email it be just a whole lot of code

Figure 32: Original text and text after pre-processing which removed symbols and tags, and applied lemmatization.

3.2 Create the Word2Vec model

I used Gensim[67], a package of python that helped me easy to create a Word2Vec model. Since I have cleaning data, before I input data into a machine, I have to split data into two parts, training data and testing data. The ratio of training and testing data is usually 8:2, but it is adjustable. Because Word2Vec model uses CBOW and Skip-Gram methods, I did not remove stop words in the data cleaning process which makes the predicted word has context relation.

```
In [13]: model = Word2Vec(size=300,min_count=10,sg=1,window=7,workers=8)

In [14]: model.build_vocab([x.words for x in tqdm(x_train)])
```

Figure 33: Word2vec training example. I set up the dimensionality of each word is 300 and count the word that appears more than at least 10 times, using the skip-gram algorithm and setting window size equals to 7.

In Figure 33, 'size' is the value of dimensionality of word embedding I defined, 'min count' means only taking in count those words whose appearance

frequency is more than 10 times, 'sg' means which training do you choose 1 is Skip-Gram and 0 is CBOW (default is 0), 'window' is the distance between current word and predicted word, and 'workers' represents how many training processes in parallel. Figure 34 shows the model creating process. In (b), we find out there are total 15,739,927 words in the training set. After removing duplicated words and low-frequency words, only 26065 distinct words in the model. Next, we begin to train the Word2Vec model.

Figure 35 shows the result after the model is trained. I set up for 30 epochs (for default, the value can be changed) and each epoch takes around 60s. Figure 37 shows the most similar word of 'love'. It will show the top 10 most similar word and its value. I save the trained Word2Vec for later used. If we want add more training data, we can load the model then feed new data in, the model will continue training.

```
INFO : collecting all words and their counts
INFO : PROGRESS: at sentence #0, processed 0 words, keeping 0 word types
INFO : PROGRESS: at sentence #10000, processed 123801 words, keeping 12797 word types
INFO : PROGRESS: at sentence #20000, processed 247861 words, keeping 20208 word types
```

(a) begin training

```
INFO : PROGRESS: at sentence #1270000, processed 15739927 words, keeping 348053 word types
INFO : collected 350033 word types from a corpus of 15863353 raw words and 1280000 sentence

INFO : Loading a fresh vocabulary
INFO : effective_min_count=10 retains 26065 unique words (7% of original 350033, drops 3239

INFO : effective_min_count=10 leaves 15354025 word corpus (96% of original 15863353, drops

INFO : deleting the raw counts dictionary of 350033 items
INFO : sample=0.001 downsamples 59 most-common words
INFO : downsampling leaves estimated 11359785 word corpus (74.0% of prior 15354025)
INFO : estimated required memory for 26065 words and 300 dimensions: 75588500 bytes
INFO : resetting layer weights
```

(b) remove duplicated words and low frequency words

Figure 34: Preprocessing data to creat a Word2Vec model.

```
2019-02-18 07:42:25,244 : INFO : EPOCH - 30 : training on 15863353 raw words (11359977 effective words) took 58.6s, 1
93728 effective words/s
2019-02-18 07:42:25,245 : INFO : training on a 475900590 raw words (340795728 effective words) took 1776.3s, 191859 e
ffective words/s
(340795728, 475900590)
```

Figure 35: Result of the Word2Vec model.

```
In [19]: model.wv.most_similar('love')

         2019-02-18 07:45:44,628 : INFO : precomputing L2-norms of word weight vectors

Out[19]: [('luv', 0.5983983278274536),
          ('lt', 0.5222497582435608),
          ('adore', 0.5128118395805359),
          ('loooove', 0.5038285851478577),
          ('lovee', 0.46899932622909546),
          ('loveeeeeeee', 0.44033411145210266),
          ('loveeeee', 0.43830278515815735),
          ('loveeeeeee', 0.4362744987010956),
          ('loveee', 0.4255843460559845),
          ('looooooooove', 0.4251766800880432)]
```

Figure 36: word similarity

```
model['love']

array([  1.03522941e-01,  -2.75688320e-02,   9.79272574e-02,
         2.04180870e-02,   7.36742988e-02,  -3.19981724e-02,
        -9.77809131e-02,   1.43957920e-02,  -2.39400193e-01,
         2.00744286e-01,  -2.88264930e-01,  -7.51685649e-02,
        -9.19567719e-02,  -1.41736552e-01,   3.66962552e-02,
         1.79437608e-01,  -6.21530274e-03,   1.80092886e-01,
         2.24816916e-03,  -1.03637554e-01,   8.49440992e-02,
        -5.93394153e-02,  -2.44710132e-01,   1.50057718e-01,
         2.84465373e-01,   4.51038294e-02,  -2.03782141e-01,
         1.70256972e-01,  -5.58725335e-02,   3.01168468e-02,
         1.69024765e-01,   8.87067020e-02,  -4.90630120e-02,
        -1.05610946e-02,   2.72082593e-02,  -2.97135413e-02,
         2.62976512e-02,  -3.25372130e-01,   8.92339796e-02,
         1.26423299e-01,   1.34793818e-01,  -2.93269694e-01,
        -2.22825721e-01,   2.16186896e-01,  -1.46561116e-01,
         4.67343023e-03,  -6.46033660e-02,   1.13902315e-01,
         1.27675056e-01,  -1.28569111e-01,  -8.32659602e-02,
         1.87193193e-02,  -6.94644228e-02,   1.80532038e-02,
        -1.10170625e-01,   2.42077440e-01,  -1.79106951e-01,
```

Figure 37: The vector of word 'love' (only shows partial of the vector). The dimension is 300

```
In [22]: len(model['love'])

Out[22]: 300

In [23]: model.similarity('love','hate')

Out[23]: 0.3401171
```

Figure 38: Word dimensions and similar score of two words. The upper half shows the length (dimensionality) of the word which is 300 (I defined). The lower half shows the similarity between the word 'love' and 'hate'. The higher value means that they have the higher similarity

3.3 ANN implementation

In this section, I create two different ANN to implement sentiment analysis on tweets. Because I saved the Word2Vec model after I trained it, the first thing is to load the model again. After that, I load the 'clean text' and 'sentiment' from a clean data file. As same as previous, I split the data into training and testing data set. Here, I set the random_state to a specific number. Sklearn provides a function that helps me easily split data into two parts. The value of random_state is the random seed to split data randomly. If I don't use this function, we have to shuffle the data set first and split them into two part. Here, x_train means 'clean text' in training data, and y_train mean the labeled data of x_train which is the 'sentiment'.

```
In [9]: x_train,x_test,y_train,y_test = train_test_split(data.clean_text.astype(str),data.sentiment,test_size=0.2,random_state
        =42)
```

Figure 39: Splitting data set into two parts

This function ensures that if the random state value is the same, then it will generate the same splitting training and testing data. Next step is tokenized sentences (training set).

After that, I will transform our sentences into vector format depending on a word dictionary. For example, there are two sentences:(1) Today is so hot (2) Today is hot and I like it. The word dictionary is {(Today,2),(is,2),(hot,2),(so,1),(and,1),(I,1),(like,1),(it,1)} and word index start from 1(0 is reserved for padding). After tokenized, sentence (1) becomes [1,2,4,3] and sentence (2) becomes [1,2,3,5,6,7,8].

```
In [13]: tokenizer = Tokenizer()
         tokenizer.fit_on_texts(x_train)
         vocab_size = len(tokenizer.word_index) + 1
         print("Total words", vocab_size)

         Total words 349164
```

Figure 40: Build a word dictionary(tokenized)

A problem while using the machine learning approach, because it is a math function, it only accepts the same and fixed size data (or same dimensionality). So I use pad_sequence function to padding at the end of sentence.

(or padding in front of sentences, it is optional) to ensure every sentence has the same length with padding 0. Also, the original data set shows 0 as a negative emotion and 4 as a positive emotion , so data in y_train is not continuous. We use LabelEncoder() function to change the value 0 as negative and 1 as positive.

```
In [14]: x_train_pad = pad_sequences(tokenizer.texts_to_sequences(x_train),maxlen=42,padding='post',truncating='post')
         x_test_pad = pad_sequences(tokenizer.texts_to_sequences(x_test),maxlen=42,padding='post',truncating='post')
         encoder = LabelEncoder()
         encoder.fit(y_train)
```

Figure 41: Padding 0 and convert training and test sentences to fix size array, encode sentiment in training and test set with negative as 0 and positive as 1 respectively

Now, I am ready create a ANN. First, I need an embedding matrix covert each word by using word vector as figure 42 shows for later used in the embedding layer.

```
In [8]: embedding_matrix = np.zeros((vocab_size,300))
        for word,i in tokenizer.word_index.items():
            if word in model:
                embedding_matrix[i] = model[word]

        print(embedding_matrix.shape)

        (349164, 300)
```

Figure 42: Embedding matrix as input. In figure 40, we know there are total 349,164 words and each word is 300 dimensions. Therefore, after embedding, the shape should be ('word size' , 'dimensionality of word vector')

In figure 43, in line [9], I load the word vector matrix into an embedding layer as an input of the network. The parameter 'trainable' = False means this embedding layer cannot be trained. Output layer I use 'sigmoid' function as a SoftMax layer output. This ANN is just a fully connected network without the hidden layer. The structure shows in In[10].

```
In [9]: embedding_layer = Embedding(vocab_size,300,weights=[embedding_matrix],input_length=42,trainable=False)

In [10]: seq_model = Sequential()
         seq_model.add(embedding_layer)
         seq_model.add(Flatten())
         seq_model.add(Dense(1,activation='sigmoid'))
         seq_model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
         print(seq_model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 42, 300)           104749200
_________________________________________________________________
flatten_1 (Flatten)          (None, 12600)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 12601
=================================================================
Total params: 104,761,801
Trainable params: 12,601
Non-trainable params: 104,749,200
_________________________________________________________________
None
```

Figure 43: First ANN structure without hidden layers.

While a model has been created, the model has to be compiled. There are a few parameters in the compile command. The first is the optimization function. Since gradient descent is widely used to optimize the model, some faster optimizations have been created. Here, I use Adam optimization (adaptive moment estimation)[35]. It is now the default optimizer of Tensorflow which has some features such as computational efficiency, it is suitable to solve sparse matrix and problems with amount of noises, and updating step does not relate to gradient[35]. The loss function that I choose is 'binary cross-entropy' which is commonly used to measure how well a machine in classification[19]. Finally, the estimated function I choose to evaluate the 'accuracy'.

Here, I set up two callback functions. One is 'ReduceLROnPlateau' and here, I used it to tweak the learning rate by monitoring validation loss. There are many parameters adjustable in Keras. I only use 'patient' and 'cooldown'. Parameter 'patient' means how many epochs have passed and the performance of the model does not increase, it will trigger the action to reduce learning rate. Parameter 'cooldown' means while learning rate has decreased, after how many epochs will trigger again. The other callback function is 'EarlyStopping'. If continue training a model that causes the performance of a model begins to decrease, the model might be overfitting or learning rate is too large which causes the model cannot converge. I use it to monitor the value of validation accuracy. Parameter 'min_delta' is a threshold and if the increasing value is greater than this threshold, it is considered

as an improvement, otherwise is not. Parameter 'patient' tolerate how many epochs does not have an improvement.

After everything is all set, I use the command in Figure 44 to train data and Figure 45 shows the progress. In Figure 44, batch size will affect the training speed and result. Batch size represents how many data will input to the model per time. The bigger size of a batch will easier get convergence, but it will increase the memory usage. Conversely, the smaller size of a batch reduce the usage of memory, but it is hard to converge the model. That is because every time when the model needs to tweak parameters, it cannot ensure the direction of the gradient. A small batch size which means parameters will tweak more times.

I save the training process into the variable 'history' and then we evaluate the model after training shows in Figure 46.

```
In [ ]: history = seq_model.fit(x_train_pad,y_train,batch_size=128,validation_split=0.1,epochs =20,verbose=1,callbacks=callbacks
```

Figure 44: Start to train a model and save training progress into history

```
Train on 1149507 samples, validate on 127724 samples
Epoch 1/20
1149507/1149507 [==============================] - 31s 27us/step - loss: 0.5024 - acc: 0.7655 - val_loss: 0.4976 - va
l_acc: 0.7697
Epoch 2/20
1149507/1149507 [==============================] - 25s 22us/step - loss: 0.4944 - acc: 0.7709 - val_loss: 0.4971 - va
l_acc: 0.7691
Epoch 3/20
1149507/1149507 [==============================] - 26s 23us/step - loss: 0.4941 - acc: 0.7709 - val_loss: 0.4988 - va
l_acc: 0.7689
Epoch 4/20
1149507/1149507 [==============================] - 25s 22us/step - loss: 0.4940 - acc: 0.7710 - val_loss: 0.4974 - va
l_acc: 0.7707
Epoch 5/20
1149507/1149507 [==============================] - 26s 22us/step - loss: 0.4940 - acc: 0.7713 - val_loss: 0.4982 - va
l_acc: 0.7692
```

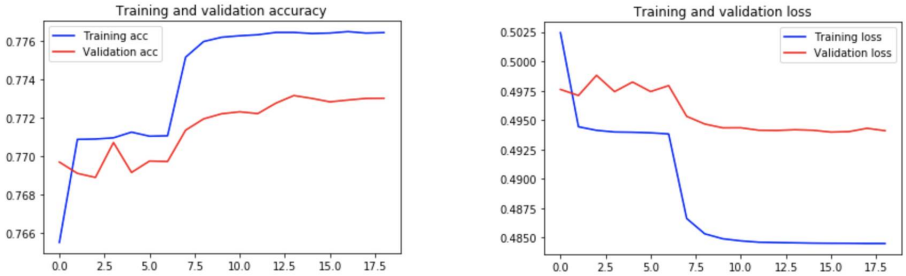
Figure 45: Training progress. Left figure is the training accuracy and validation accuracy. Right figure is the training loss and validation loss.

```
%%time
score = seq_model.evaluate(x_test_pad, y_test, batch_size=1000)
print()
print("ACCURACY:",score[1])
print("LOSS:",score[0])
```

```
319308/319308 [==============================] - 1s 2us/step

ACCURACY: 0.772473598512
LOSS: 0.493024488995
CPU times: user 960 ms, sys: 173 ms, total: 1.13 s
Wall time: 628 ms
```

Figure 46: Evaluate the model with test data



(a) training accuracy (b) training loss

Figure 47: Training progress

Plotting the training progress shows in Figure 47 and a confusion matrix in Figure 48.

Figure 49 shows the F1-score and accurate rate. Now, we can enter our own sentences to the machine and make a prediction (in Figure 50). In this model, if a value is greater than 0.5 then it is considered as 'Positive', and less than 0.5 is considered as 'Negative'. Now, we have already created a sentiment classifier, even it is not so precise in our sentences. The first ANN model gave me a 77.24% accurate rate. I will try a different ANN model.

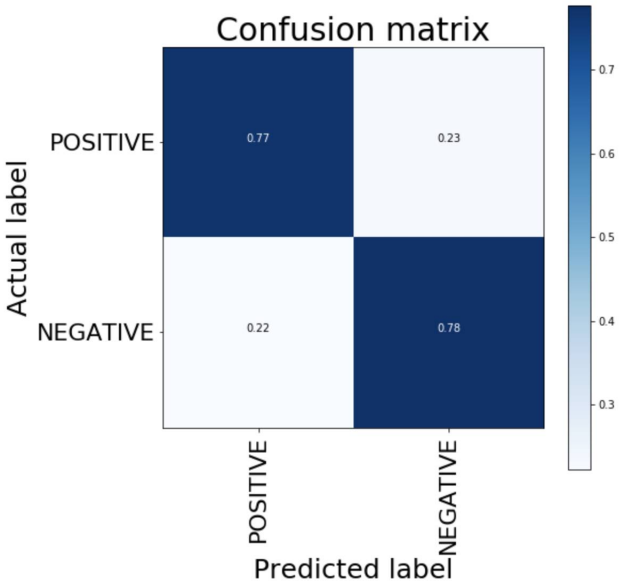


Figure 48: Confusion matrix of first ANN

```
In [24]: print(classification_report(y_test_1d, y_pred_1d))
                precision    recall  f1-score   support

      NEGATIVE       0.77      0.77      0.77    159499
      POSITIVE       0.77      0.78      0.77    159809

avg / total       0.77      0.77      0.77    319308

In [25]: accuracy_score(y_test_1d, y_pred_1d)
Out[25]: 0.77247359915817959
```

Figure 49: F1 score and accuracy of first ANN. (77.24% accuracy)

```
predict_sentence("The dog is adorable.")
```

```
{'elapsed_time': 0.0024046897888183594,
 'label': 'POSITIVE',
 'score': 0.7236402034759521}
```

```
predict_sentence("Tedious work is annoying.")
```

```
{'elapsed_time': 0.0022077560424804688,
 'label': 'POSITIVE',
 'score': 0.6198058128356934}
```

```
predict_sentence("I hate the raining day.")
```

```
{'elapsed_time': 0.002084493637084961,
 'label': 'NEGATIVE',
 'score': 0.06196475401520729}
```

```
predict_sentence("i don't know what i'm doing")
```

```
{'elapsed_time': 0.0021533966064453125,
 'label': 'POSITIVE',
 'score': 0.5383113622665405}
```

Figure 50: Test sentences example. I entered four sentences to evaluate the model.

I create the second ANN model with one hidden layer. The structure shows in figure 51. Figure 52 and figure 53 show scores and training processes respectively.

```
seq_model_2 = Sequential()
seq_model_2.add(embedding_layer)
seq_model_2.add(Dense(300,activation='relu'))
seq_model_2.add(Flatten())
seq_model_2.add(Dense(1,activation='sigmoid'))
seq_model_2.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
print(seq_model_2.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 42, 300)           104749200
_________________________________________________________________
dense_2 (Dense)              (None, 42, 300)           90300
_________________________________________________________________
flatten_2 (Flatten)          (None, 12600)             0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 12601
=================================================================
Total params: 104,852,101
Trainable params: 102,901
Non-trainable params: 104,749,200
```

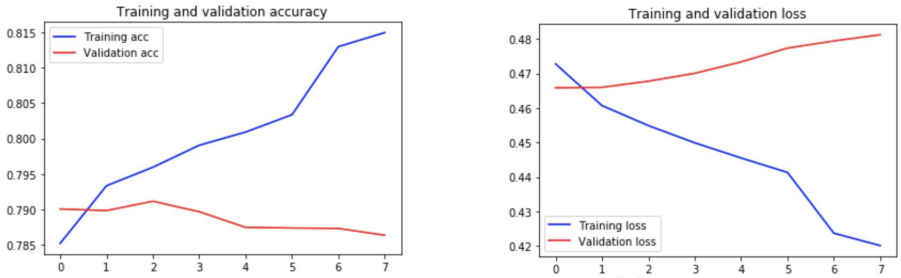
Figure 51: Second ANN with one hidden layer

```
%%time
score = seq_model_2.evaluate(x_test_pad, y_test, batch_size=1000)
print()
print("ACCURACY:",score[1])
print("LOSS:",score[0])
```

```
319308/319308 [==============================] - 1s 3us/step

ACCURACY: 0.785492376607
LOSS: 0.478732090923
CPU times: user 921 ms, sys: 275 ms, total: 1.2 s
Wall time: 1.12 s
```

Figure 52: Evaluate second ANN model with test data and get a 78.54% accuracy.



(a) training accuracy (b) training loss

Figure 53: Second ANN model training progress

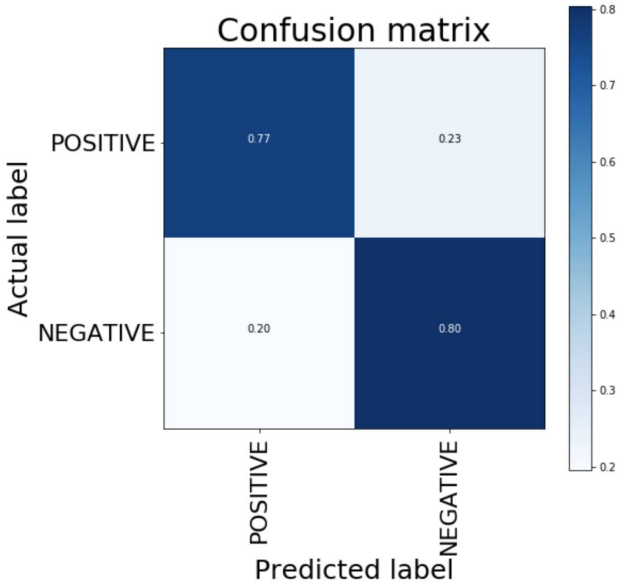


Figure 54: Confusion matrix of second ANN

```
print(classification_report(y_test_1d, y_pred_1d))
accuracy_score(y_test_1d, y_pred_1d)
```

```
             precision    recall  f1-score   support

   NEGATIVE       0.80      0.77      0.78    159499
   POSITIVE       0.78      0.80      0.79    159809

avg / total       0.79      0.79      0.79    319308

0.78549237726583743
```

Figure 55: F1 score and accuracy of second ANN

```
predict_sentence("The dog is adorable.")
```

```
{'elapsed_time': 0.002506256103515625,
 'label': 'POSITIVE',
 'score': 0.814437747001648}
```

```
predict_sentence("Tedious work is annoying.")
```

```
{'elapsed_time': 0.002063751220703125,
 'label': 'NEGATIVE',
 'score': 0.28556889295578003}
```

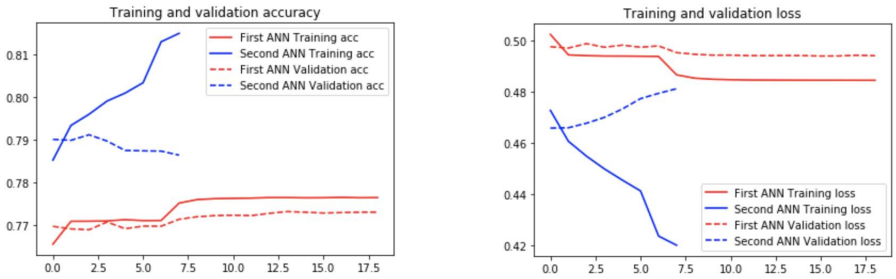
```
predict_sentence("I hate the raining day.")
```

```
{'elapsed_time': 0.0021820068359375,
 'label': 'NEGATIVE',
 'score': 0.02276407741010189}
```

```
predict_sentence("i don't know what i'm doing")
```

```
{'elapsed_time': 0.0021758079528808594,
 'label': 'POSITIVE',
 'score': 0.5160800218582153}
```

Figure 56: Test sentences on second ANN



(a) training accuracy (b) training loss

Figure 57: Compare two ANN model of training accuracy and loss

As we see in Figure 55. After adding one hidden layer, the accuracy increase from 77.24% to 78.54%. Evaluating our own sentences, the second model classifies sentence(2) as negative which the result is different from the first model. It seems the second has a better ability of classification in our own sentences. Sentence 1 and sentence 2 get enough score to classify

as positive and negative respectively, which is better than the first model classifying sentence 1 and 2 more closed.

3.4 CNN Model

Even though CNN is widely used in image recognition, CNN also has the ability in processing natural language[34].

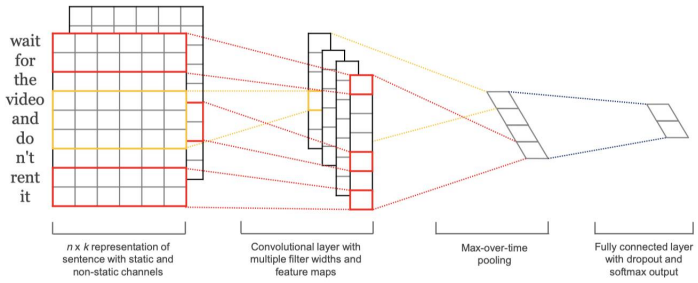


Figure 58: Showing how CNN model works in natural language [34]

Traditionally, an image is a two dimensional matrix. In Keras, we use 'Conv2D' to process convolution in images, yet in natural language, we need use Conv1D. While creating a CNN model, most parts are as same as creating an ANN model including using the same Word2Vec model, splitting same data set into training and testing data, and identical padding sentences and word embedding matrix excepts adding 'MaxPooling' and 'Dropout' layers in CNN. There was an example in "A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification"[87] showing in figure 59. In this example, each word is 5 dimensions, and convolution with kernel size 2, 3, and 4 which represents as bi-grams,tri-grams, and 4-grams in natural language. Each size of the kernel has two, so there are total 6 filters. It will generate 6 different vectors and concatenate them together after applying 'Maxpooling' of each vector. Finally, it will get a probability of two categories.

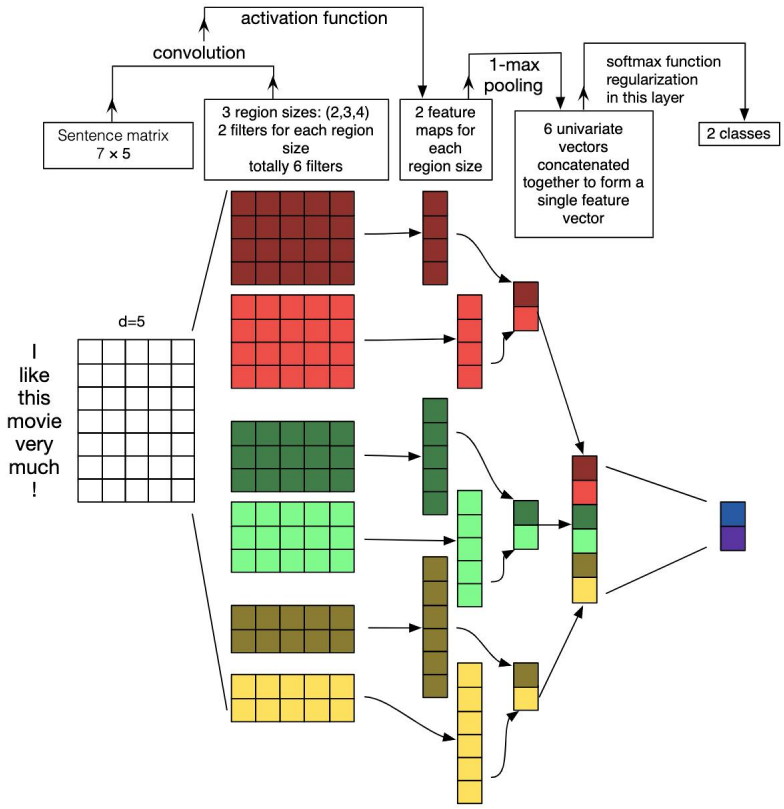


Figure 59: A three channels CNN[87]

My first CNN model structure as shown in Figure 60 and Figure 61. Here, I add a dropout layer in each channel and use 'relu' as an activation function. In my model, the kernel size of every convolution layer represent n-grams features and every convolution layer follows with one dropout layer. In the embedding layer, word vector size is 300. Before the output layer, there is a merged layer concatenating three vectors after convolution, and dense_layer is a fully connected layer. Output layer uses 'sigmoid' activation function producing the probability of classifier. Every epoch average trains in 930 seconds. This machine gives me an around 80.42% accuracy.

After this, I tried another CNN model which is shown in Figure 62 and Figure 63. The second model gave me almost the same accuracy 80.03%. When I reviewed the training process, both models received increasing training accuracy which was around 90%, but validation accuracy was decreasing. It seems two model are overfitting and I need a sophisticated model or more

training data. Changing three same inputs and applying convolution layers individually then merged together to one input applying three convolution layers does not increase the performance of the machine. Comparison of training process is shown in Figure 64.

```
input_1 = Input(shape=(42,))
embedding_1 = Embedding(vocab_size,300)(input_1)
conv_1 = Conv1D(filters=16,kernel_size=4,activation='relu')(embedding_1)
drop_1 = Dropout(0.5)(conv_1)
pool_1 = MaxPooling1D(pool_size=2)(drop_1)
flat_1 = Flatten()(pool_1)

input_2 = Input(shape=(42,))
embedding_2 = Embedding(vocab_size,300)(input_2)
conv_2 = Conv1D(filters=16,kernel_size=6,activation='relu')(embedding_2)
drop_2 = Dropout(0.5)(conv_2)
pool_2 = MaxPooling1D(pool_size=2)(drop_2)
flat_2 = Flatten()(pool_2)

input_3 = Input(shape=(42,))
embedding_3 = Embedding(vocab_size,300)(input_3)
conv_3 = Conv1D(filters=16,kernel_size=8,activation='relu')(embedding_3)
drop_3 = Dropout(0.5)(conv_3)
pool_3 = MaxPooling1D(pool_size=2)(drop_3)
flat_3 = Flatten()(pool_3)

merged = concatenate([flat_1, flat_2, flat_3])
dense_layer = Dense(10,activation='relu')(merged)
output_layer = Dense(1, activation= 'sigmoid' )(dense_layer)
textCNN = Model(inputs = [input_1,input_2,input_3],outputs=output_layer)
textCNN.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
textCNN.summary()
```

Figure 60: First CNN comprises three individual inputs and convolution layers with kernel size = 4,6,8

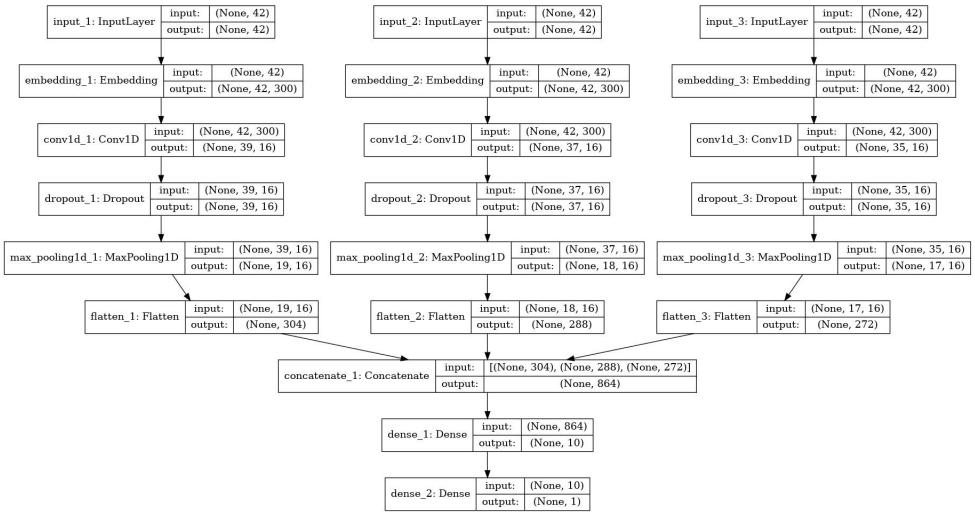


Figure 61: Visualization of first CNN model

```
input_2 = Input(shape=(42,))
embedding_2 = Embedding(vocab_size,300)(input_2)

conv_1 = Conv1D(filters=16,kernel_size=4,activation='relu')(embedding_2)
pool_1 = MaxPooling1D(pool_size=2)(conv_1)
flat_1 = Flatten()(pool_1)

conv_2 = Conv1D(filters=16,kernel_size=6,activation='relu')(embedding_2)
pool_2 = MaxPooling1D(pool_size=2)(conv_2)
flat_2 = Flatten()(pool_2)

conv_3 = Conv1D(filters=16,kernel_size=8,activation='relu')(embedding_2)
pool_3 = MaxPooling1D(pool_size=2)(conv_3)
flat_3 = Flatten()(pool_3)

merged = concatenate([flat_1, flat_2, flat_3])
drop_out = Dropout(0.5)(merged)
output_layer = Dense(1, activation= 'sigmoid' )(drop_out)
textCNN_2 = Model(inputs = [input_2],outputs=output_layer)
textCNN_2.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
textCNN_2.summary()
```

Figure 62: Second CNN comprises single input with three channels and filter sizes = 4,6,8

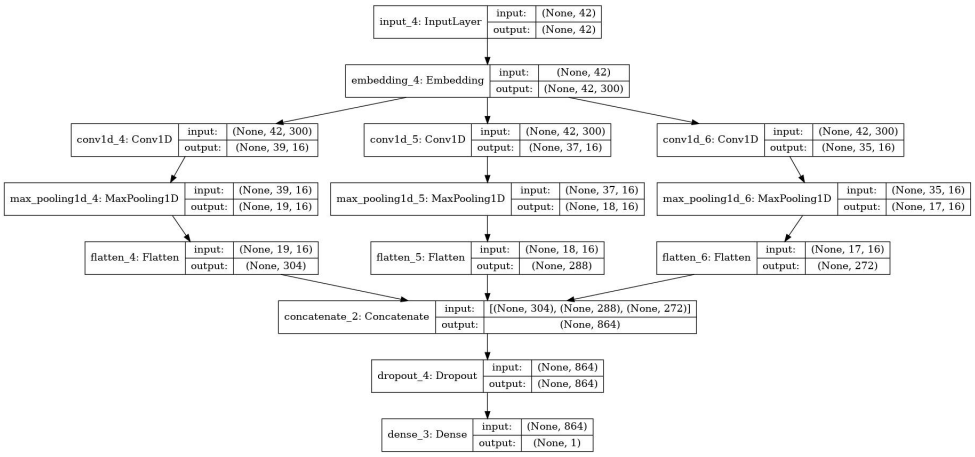
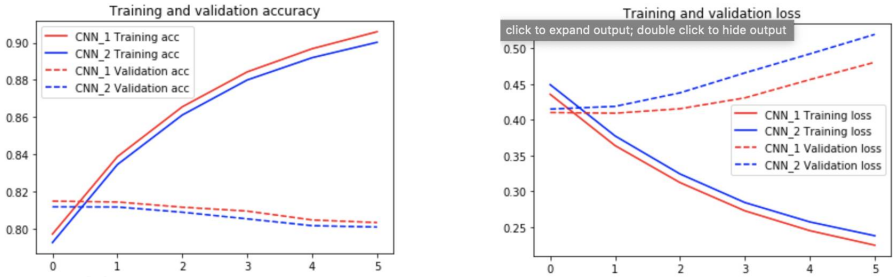


Figure 63: Visualization of second CNN model



(a) training accuracy (b) training loss

Figure 64: Comparison of two CNN of training accuracy and loss(both overfitting)

3.5 LSTM model

The last model I tried was LSTM. Inside of a LSTM cell is intricate, but Keras already provides a high-level API for usage. The structure of LSTM model shows in figure 65. I got the result that pointed the LSTM get the highest accuracy which is around 82.62% higher than CNN and ANN. While training a LSTM model, it took about 640 seconds per epochs. It's longer than ANN but shorter than CNN. Figure 66 shows training progress, Figure 67 is the confusion matrix, and Figure 68 is the result. For now, the LSTM model has the best performance in sentiment classification.

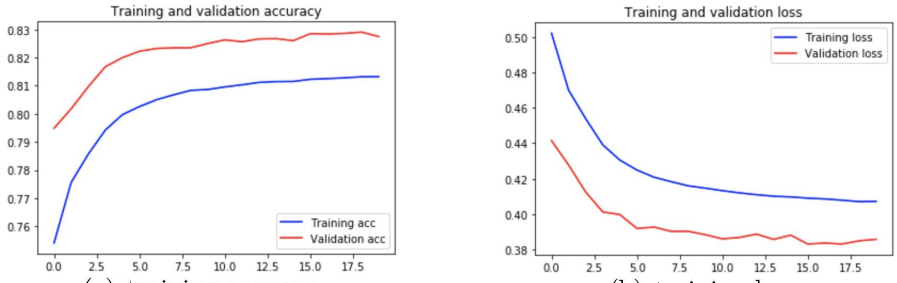
```
LSTM_model = Sequential()
LSTM_model.add(embedding_layer)
LSTM_model.add(Dropout(0.5))
LSTM_model.add(LSTM(100,dropout=0.2,recurrent_dropout=0.2))
LSTM_model.add(Dense(1,activation='sigmoid'))
LSTM_model.summary()

LSTM_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

callbacks = [ ReduceLROnPlateau(monitor='val_loss', patience=5, cooldown=0),
              EarlyStopping(monitor='val_acc', min_delta=1e-4, patience=5)]
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 42, 300)           104749200
_________________________________________________________________
dropout_1 (Dropout)          (None, 42, 300)           0
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               160400
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 104,909,701
Trainable params: 160,501
Non-trainable params: 104,749,200
```

Figure 65: LSTM model with one dropout layer and internal dropout of LSTM cell (neurons=100).



(a) training accuracy (b) training loss

Figure 66: LSTM model training progress

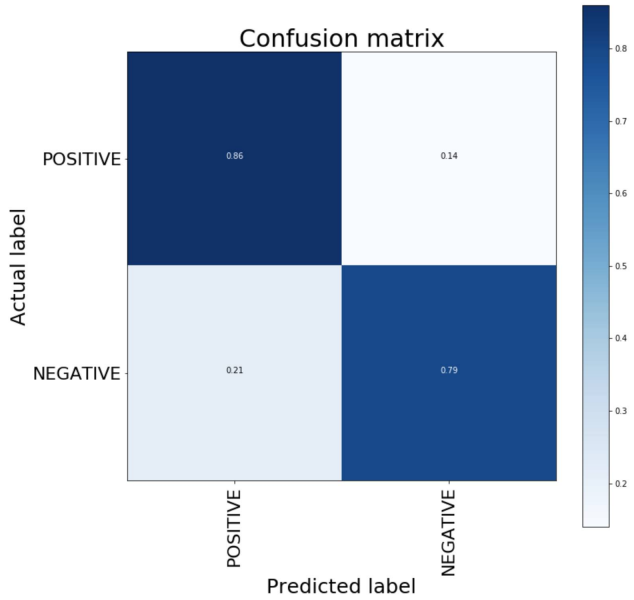


Figure 67: confusion matrix of LSTM

```
print(classification_report(y_test_1d, y_pred_1d))
```

	precision	recall	f1-score	support
NEGATIVE	0.81	0.86	0.83	159499
POSITIVE	0.85	0.79	0.82	159809
avg / total	0.83	0.83	0.83	319308

```
accuracy_score(y_test_1d, y_pred_1d)
```

```
0.82629937239279938
```

Figure 68: Precision,recall,f1 score, and accuracy (82.62%) of LSTM model

3.6 Use word embedding without pre-trained

According to observations above, now, using a distributed representation in machine learning achieves the classification accuracy rate is between 77.24% and 82.62%. I wonder how the performance will be if I use one hot representation in machine learning. Unfortunately, while I tried to transform

sentences into one hot encoding, there came out a memory problem. Because the vocabulary size is almost 408,967 in the whole data set, if using one hot encoding, each word will be a [1, 408,967] dimensions. My machine did not have enough memory to accommodate the data set using one hot encoding. Therefore, here, I use a model without pre-trained word embedding instead. Each word represented as a distinct value in the data set instead of one large sparse array for every word. The only major difference between this and Word2Vec is the value in the sentence is just described as the word index which is the position of the word dictionary. We cannot identify the relation between two words. Therefore, the machine only learns from training data based on the position of words. I use the same two ANN models to test this encoding.

Not surprising, the first ANN without a hidden layer got a 69.32% accuracy rate and recognized all custom sentences as negative. However, it is amazing with one hidden layer of ANN, this type of word embedding get a 78.35% accuracy rate which is only slightly worse than using Word2Vec embedding.

After I tried to build ANN, the next step was to build CNN. Two CNN model structures are as same as previous in section 3.4 which gave similar accuracy and both were overfitting again. I got 80.39% and 79.94% accuracy rate.

The last model I tried was LSTM. While I was training the model, the accuracy rate was between 49% and 50% without any improvement for every epoch which was very low, and loss rate was around 69% which was very high that made me feel confused about the model I created. Was that model too complicated to fit in the data? Then I tried a second LSTM with less complexity shown in Figure 69 and it gave me a higher accuracy is about 77.23%. It seems a little weird, so I try the first model again and this time I adjust the batch size from 128 to 1024.

```
LSTM_model = Sequential()
LSTM_model.add(embedding_layer)
LSTM_model.add(Dropout(0.5))
LSTM_model.add(LSTM(100,dropout=0.2,recurrent_dropout=0.2))
LSTM_model.add(Dense(1,activation='sigmoid'))
LSTM_model.summary()

LSTM_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

callbacks = [ ReduceLROnPlateau(monitor='val_loss', patience=5, cooldown=0),
              EarlyStopping(monitor='val_acc', min_delta=1e-4, patience=5)]
```

(a) Fisrt LSTM

```
LSTM_model_2 = Sequential()
LSTM_model_2.add(embedding_layer)
LSTM_model_2.add(LSTM(20,dropout=0.2,recurrent_dropout=0.2))
LSTM_model_2.add(Dense(1,activation='sigmoid'))
LSTM_model_2.summary()

LSTM_model_2.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

callbacks = [ ReduceLROnPlateau(monitor='val_loss', patience=5, cooldown=0),
              EarlyStopping(monitor='val_acc', min_delta=1e-4, patience=5)]
```

(b) Second LSTM

Figure 69: Training two LSTM model without pre-trained word embedding.

It seemed to become normal this time, the accuracy was raising from

53.20%, 53.85%, 57.34%, ... and finally stop at 75.75%. Then I realized maybe the problem was when I trained the model, algorithm tuned the weights by the loss function just fell into a local minimum. That's the reasonable explanation of why accuracy did not increase and the loss did not decrease. I compared two models with different batch which received the results in Table 4.

	batch size=128	batch size=1024
LSTM (100)	49.95%	75.75%
LSTM (20)	77.23%	77.59%

Table 4: Accuracies of two different LSTM with different batch size

3.7 Try different model and tweak parameters

According to sections above, I notice per-trained word embedding such as Word2Vec has a better performance than a model without pre-trained in machine learning. But can we do better? Therefore, I tried to create different models and tweak different parameters to gain a better accuracy with pre-trained word embedding. The first model I tried was a deeper CNN which with 4 layer of convolution layers and 6 dropout layers. Unfortunately, it did not give me a better performance which was 77.29% even less than the original model(around 80%).

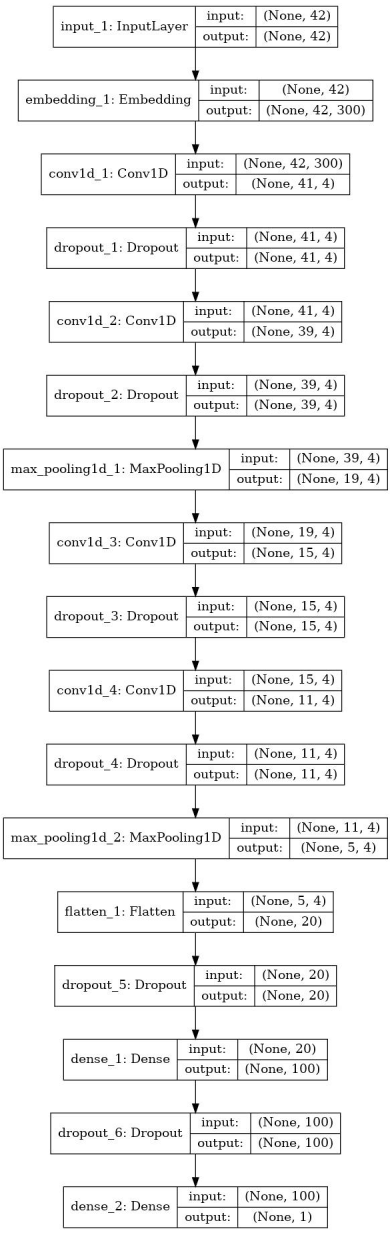


Figure 70: A three layers CNN model

The second model I tried was more neurons of LSTM and add one more dropout layer and a dense layer. This model gave me the highest accuracy, for now, I got a 83.89% accurate rate.

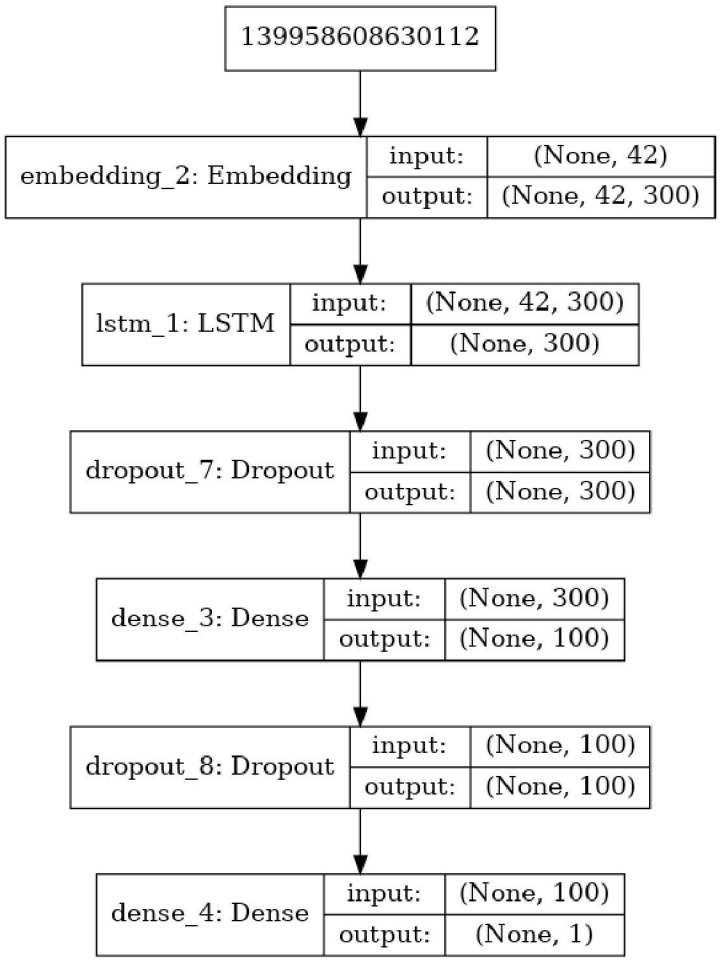


Figure 71: A more complex LSTM(83.89% accuracy) model with one more hidden layer and a dropout layer.

The third model I combined two types of neural networks which are CNN+LSTM. It gave me 79.44% accuracy rate which the performance did not improve.

model	accuracy
First ANN	77.24%
Second ANN	78.54%
First CNN	80.42%
Second CNN	80.03%
LSTM (100 neurons)	82.62%
Deeper CNN	77.29%
LSTM (300 neurons)+ dropout and one dense layer	83.89%
CNN+LSTM (100 neurons)	79.44%

Table 5: Training machines with pre-trained word embedding. The LSTM model with 300 neurons, one dropout layer, and one more dense layer gets the highest accuracy (83.89%)

model	accuracy
First ANN	69.32%
Second ANN	78.35%
First CNN	80.39%
Second CNN	79.94%
LSTM (100 neurons) batch size=128	49 50%
LSTM (20 neurons) batch size=128	77.23%
LSTM (100 neurons) batch size=1024	75.75%
LSTM (20 neurons) batch size=1024	77.59%

Table 6: Training machines without pre-trained word embedding. The first CNN model gets the highest accuracy (80.39%).

Besides, I wonder to know how the different Word2Vec model will affect the machine, so I trained other Word2Vec models include reducing dimensionality and remove stop word that gave me results in Table 5. Besides, while I trained the model, there was an optional parameter which indicated weights will be trainable or not, and I also evaluated this option with the LSTM. With trainable parameters, the performance of the model did not increase, conversely, it reduced the performance from 82.62% to 81.16%.

Finally, I tried the pre-trained Word2Vec model which is trained by Google with News which was trained with 100 billion words. However, when

I used this model, it did not give me a significant performance increasing, and the accuracy was 80.67% which was lower than my own model. I tried three traditional machine learning model which are 'Logistic Regression', 'Random Forest Tree', and 'Naive Bayes' by TF-IDF feature. They resulted in 78.27%, 73.72%, and 76.69% accuracy respectively.

	LSTM accuracy
dim (100)	80.13%
dim (300)	82.62%
dim (300) + trainable parametes	81.16%
dim (100) + remove stop-words	78.82%

Table 7: The accuracy result of a LSTM with different word2vec models. The first model set the word embedding size to 100 and second one set to 300. The last one set to 100 and remove stop words. It shows the results that removing stop words will affect the accuracy when we use the Word2Vec embedding.

	model accuracy
Logistic Regression	78.27%
Random Forest Tree	73.72%
Naive Bayes	76.69%

Table 8: Using simple machine learning approaches instead of artificial neural network.

4 Conclusion

After implementation, I found that the use of machine learning in sentiment analysis is practical. Compared to the traditional approach which are based on complex and user defined linguistic rules, it is more efficienct. Moreso, Twitter consists of informal language and includes words that have not yet been included in the modern dictionary. For example, words such as 'luv', 'looooooooove', and 'grr' that does not exist in the dictionary. Humans can recognize the meaning but computers cannot. If I use the traditional approach, I have to define the sentence structure and find the semantics of words, then analyze the sentiment. This approach is time consuming and tedious. Using the machine learning approach, I just need to grab the data and apply the data preprocessing, then I can feed data input the ANN to create a model then predict the sentiment. Additionally, with more sentences, I do not need to re-design a new model, I only need to preprocess new data (which contains input data and labeled data) and input it into the model. I do not need to create the model again, all I have to do is to perform the training process again, then the ANN will retrain itself and tune the model then generate the result.

Additionally, using distributed representation of words in sentences such as Word2Vec, I also show better performance while utilzing less computing resources than without pre-trained model. While a corpus can become massive, using a distributed representation in NLP is a better choice. However, the machine learing approach invloves knowledge across many disciplines, it is not intuitive to understand how the model works and how it is trained. Moreover, to optimize the model, it relies on experiences. In this thesis, I conclude numerous interesting points.

I discovered that the more sophisticated and the model is deeper will not always guarantee better performance. For example, in a CNN model, even we increased the number of convolution layers, it did not work better than a simple model. Also, a simple machine learning model 'Logistic Regression', with TF-IDF feature extraction, still achieved almost 78% accuracy as an ANN model. Since I only tried to evaluate how good I can classify a sentiment emotion on tweets, actually, linear regression does a great job on handling spare data set. Therefore, depending on what kind of data you are

processing, to choose the correct one is important.

I also discovered that, a CNN model performs quite well for natural language processing. It did an acceptable job in natural language, but I think it is not very appropriate in sentiment analysis on tweets for a reason. The power of convolution layer is to try to extract more features by performing the convolution, but tweets are too short and contain informal language, or maybe the language itself does not have many kinds of features.

Additionally, a LSTM model in natural language achieved the best performance. While I extended one more dense layer, it resulted in 83.94% accuracy rate. Even without a dense layer, it still achieved 82.62% accuracy. Also in an ANN network, while I added one more hidden layer, the performance had slightly increased. It shows hidden layer may increase the performance but contradict to the first point, not always the deeper neural network will get better performance.

Additionally, I also tested if more neurons would achieve better performance. The answer remains doubtful, since it depends on different situations. For example, in section 3.6 training a LSTM model without pre-trained word embedding, a LSTM model with 20 neurons got a 77.23% and 77.59% accuracy better than a model with 100 neurons which only got 49.95% and 75.75% accuracy on different batch sizes. While training a model, if we do not have too much data and features in data set, too many neurons may not be able to fit in data set that causes a overfitting. Also, in the experiments in section 3.6, we noticed that batch size will influence the training process. If the batch size is too small, while calculating the loss function, the model might not escape the local minimum and converge. It will generate the result as same as while the learning rate is too big.

In this thesis, I used a pre-trained Word2Vec model which is trained by Google with tons of words from Google News. It contains 3 millions word vectors and pretrained from roughly 100 billions words. Intuitively, the word embedding model should work much better than my own custom model, but the fact does not correspond with our first thought. In machine learning, the performance of a machine has a strong relationship with training data. Therefore, the data in the corpus is important. Google trained its model with Google News and the point is that tweets may not have so many formal

words in its data set. Even more, many words in the Google model may not appear in the training set. For example, Figure 72 shows the top 10 words that relate to the word 'love' and we can see the difference. Some words in the Google model will not apprear in the training data and some words in the training data may not show in Google model too. This kind of mismatch decreases the degree of similarity between training sentences. Thus, training a Word2Vec model within its corpus which will use in the machine learning approach that represent a better relationship between each word in training data.

```
[('luv', 0.5983983874320984),
 ('lt', 0.5222498178482056),
 ('adore', 0.5128117799758911),
 ('loooove', 0.5038286447525024),
 ('lovee', 0.46899932622909546),
 ('loveeeeeeee', 0.4403340518474579),
 ('loveeeee', 0.43830281496047974),
 ('loveeeeeee', 0.4362744688987732),
 ('loveee', 0.4255843460559845),
 ('looooooooove', 0.4251766800880432)]
```

(a) custom

```
[('loved', 0.6907792091369629),
 ('adore', 0.6816873550415039),
 ('loves', 0.661863386631012),
 ('passion', 0.6100709438323975),
 ('hate', 0.600395679473877),
 ('loving', 0.5886635780334473),
 ('Ilove', 0.5702950954437256),
 ('affection', 0.5664337873458862),
 ('undying_love', 0.5547305345535278),
 ('absolutely_adore', 0.5536840558052063)]
```

(b) google

Figure 72: Custom and Google pre-trained Word2Vec model show different result in top 10 words that are most similar to word 'love'

Across all of the implementations, I got an accurate rate between around 73% to 84%. I even tried different models to improve the performance, but it did not work for me. There may have been some specific reasons. First, the data quantity is not enough. As we know, if we feed more data into the machine, then the machine will learn on itself continuously. Second, the data quality may not be good enough. Data quality has an absolute connection to the performance of machines. If we give a machine wrong or dirty training data, then the machine cannot result in the performance as you excepted. Third, maybe my implementation of feature extraction is not good enough and that causes a machine cannot learn from these features very well.

Even tweaking hyper-parameters with a machine can get a better performance, however, tweaking parameters is a tedious and time-consuming task. Moreover, it may increase my model's performance, but it has a limitation, it won't give me a significantly abrupt increment. A better way is to review what is the problem with my model. Does it have problems with machine design or is it the data problem? In my experience, tweaking parameters within the same model will not get a significant increment of performance

than changing a model, providing more features, and cleaning training data.

Finally, increasing the number of data is a practical strategy to gain a better performance in machine learning. There are many ways to do that. The most straight forward is to grab more data as much as possible. However, in the real world, sometimes it is impractical. Getting labeled data usually needs a surprising amount of cost and time. Another way to achieve this goal is called 'data augmentation'[78]. For example, in image recognition, we can generate more images from the original data set by rotating images, sharpening images, adding some noises in the images, and resizing images. Unfortunately, data augmentation is a little harder in natural language. Traditionally, we may randomly remove some words in a sentence, replace words with synonyms randomly, and shuffle the order of a sentence to generate more training data. However, while using a Word2Vec model, replacing words with synonyms may confuse the machine because, in Word2Vec, a word's synonyms and antonyms both have a closer similarity. This may be a problem in replacing words with synonyms. Removing words randomly and reordering words are also not practical because we know machine learning has a dependency on the sequence of words. Removing words or reordering sentences will also remove this kind of feature from the data set. Getting more labeled data in natural language is hard but crucial in machine learning.

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Apoorv Agarwal, Boyi Xie, Ilia Vovsha, Owen Rambow, and Rebecca Passonneau. Sentiment analysis of twitter data. In *Proceedings of the Workshop on Language in Social Media (LSM 2011)*, pages 30–38, 2011.

[3] Akiko Aizawa. An information-theoretic perspective of tf-idf measures. *Information Processing and Management*, 39(1):45–65, 2003.

[4] Eiman Tamah Al-Shammari. Lemmatizing, stemming, and query expansion method and system, June 25 2013. US Patent 8,473,279.

[5] Muhammad Zubair Asghar, Aurangzeb Khan, Shakeel Ahmad, and Fazal Masud Kundi. A review of feature extraction in sentiment analysis. *Journal of Basic and Applied Scientific Research*, 4(3):181–186, 2014.

[6] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, 2008.

[7] Mariusz Bernacki. Principles of training multi-layer neural network using backpropagation algorithm.

[8] Pushpak Bhattacharyya. Natural language processing: A perspective from computation in presence of ambiguity, resource constraint and multilinguality. *CSI journal of computing*, 1(2):1–13, 2012.

[9] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[10] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 111–118, 2010.

[11] Michael Buckland and Fredric Gey. The relationship between recall and precision. *Journal of the American society for information science*, 45(1):12–19, 1994.

[12] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[13] Angelo Cangelosi. Evolution of communication and language using signals, symbols, and words. *IEEE Transactions on Evolutionary Computation*, 5(2):93–101, 2001.

[14] Jim X Chen. The evolution of computing: Alphago. *Computing in Science and Engineering*, 18(4):4, 2016.

[15] Laurence Danlos. The linguistic basis of text generation. In *Proceedings of the third conference on European chapter of the Association for Computational Linguistics*, pages 1–1. Association for Computational Linguistics, 1987.

[16] Geoff Dougherty. *Pattern recognition and classification: an introduction.* Springer Science and Business Media, 2012.

[17] Bradley J Erickson, Panagiotis Korfiatis, Zeynettin Akkus, and Timothy L Kline. Machine learning for medical imaging. *Radiographics*, 37(2):505–515, 2017.

[18] Mark Gebhart, Stephen W Keckler, Brucek Khailany, Ronny Krashinsky, and William J Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 96–106. IEEE, 2012.

[19] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[20] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision.

[21] François Guély and Patrick Siarry. Gradient descent method for optimizing various fuzzy rule bases. In *[Proceedings 1993] Second IEEE International Conference on Fuzzy Systems*, pages 1241–1246. IEEE, 1993.

[22] Emma Haddi, Xiaohui Liu, and Yong Shi. The role of text pre-processing in sentiment analysis. *Procedia Computer Science*, 17:26–32, 2013.

[23] Simon Haykin. *Neural networks*, volume 2. Prentice hall New York, 1994.

[24] Geoffrey E Hinton. Distributed representations. 1984.

[25] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[26] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[27] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] Hans Henrich Hock and Brian D Joseph. *Language history, language change, and language relationship: An introduction to historical and comparative linguistics*, volume 218. Walter de Gruyter, 2009.

[30] Anna Huang. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, volume 4, pages 9–56, 2008.

[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[32] Anil K Jain, Jianchang Mao, and KM Mohiuddin. Artificial neural networks: A tutorial. *Computer*, (3):31–44, 1996.

[33] Aditya Joshi, AR Balamurali, and Pushpak Bhattacharyya. A fall-back strategy for sentiment analysis in hindi: a case study. *Proceedings of the 8th ICON*, 2010.

[34] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[35] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[36] Efthymios Kouloumpis, Theresa Wilson, and Johanna Moore. Twitter sentiment analysis: The good the bad and the omg! In *Fifth International AAAI conference on weblogs and social media*, 2011.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[38] Leah S Larkey, Lisa Ballesteros, and Margaret E Connell. Improving stemming for arabic information retrieval: light stemming and co-occurrence analysis. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–282. ACM, 2002.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[41] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient descent only converges to minimizers. In *Conference on Learning Theory*, pages 1246–1257, 2016.

[42] Edda Leopold and Jörg Kindermann. Text categorization with support vector machines. how to represent texts in input space? *Machine Learning*, 46(1-3):423–444, 2002.

[43] Baoli Li and Liping Han. Distance weighted cosine similarity measure for text classification. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 611–618. Springer, 2013.

[44] Elizabeth D Liddy. Natural language processing. 2001.

[45] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[46] Bing Liu. Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies*, 5(1):1–167, 2012.

[47] Haibin Liu, Tom Christiansen, William A Baumgartner, and Karin Verspoor. Biolemmatizer: a lemmatization tool for morphological processing of biomedical text. *Journal of biomedical semantics*, 3(1):3, 2012.

[48] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

[49] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[50] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[51] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[52] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[53] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*, 2013.

[54] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[55] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[56] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[57] Martin A Nowak and David C Krakauer. The evolution of language. *Proceedings of the National Academy of Sciences*, 96(14):8028–8033, 1999.

[58] Franz Josef Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 160–167. Association for Computational Linguistics, 2003.

[59] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[60] Nicholas Ostler. *Empires of the word: A language history of the world*. HarperCollins New York, 2005.

[61] Mark Pagel and Andrew Meade. The deep history of the number words. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 373(1740):20160517, 2018.

[62] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREc*, volume 10, pages 1320–1326, 2010.

[63] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*, page 271. Association for Computational Linguistics, 2004.

[64] Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, 2(1–2):1–135, 2008.

[65] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 79–86. Association for Computational Linguistics, 2002.

[66] Juan Ramos et al. Using tf-idf to determine word relevance in document queries.

[67] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[68] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. *Nature neuroscience*, 2(11):1019, 1999.

[69] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[70] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[71] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[72] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[73] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[74] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[75] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[76] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[77] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[78] Martin A Tanner and Wing Hung Wong. The calculation of posterior distributions by data augmentation. *Journal of the American statistical Association*, 82(398):528–540, 1987.

[79] Jack V Tu. Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes. *Journal of clinical epidemiology*, 49(11):1225–1231, 1996.

[80] Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *International Work-Conference on Artificial Neural Networks*, pages 758–770. Springer, 2005.

[81] Xin Wang, Yuanchao Liu, SUN Chengjie, Baoxun Wang, and Xiaolong Wang. Predicting polarities of tweets by composing word embeddings with long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1343–1353, 2015.

[82] Dean Weber. Network interactive user interface using speech recognition and natural language processing, March 11 2003. US Patent 6,532,444.

[83] Peter Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[84] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[85] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification.

In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, 2016.

[86] Wen Zhang, Taketoshi Yoshida, and Xijin Tang. A comparative study of tf* idf, lsi and multi-words for text classification. *Expert Systems with Applications*, 38(3):2758–2765, 2011.

[87] Ye Zhang and Byron Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*, 2015.

[88] Jacek M Zurada. *Introduction to artificial neural systems*, volume 8. West publishing company St. Paul, 1992.