Tweets sentiment analysis using convolutional neural network (CNN)

Alberto Gabriel Buftea

October 28, 2020

Contents

List of Figures

Abstract

As social media information becomes more and more important in the new digital era, companies stays more and more tuned in for information or feedback from customers which could affect their interests. A good example of the importance of this are the tweets published by the owner of Tesla, Elon Musk, that immediately affected the stock price after their publishing. In a similar way, tweets published by the U.S. president Donald Trump in relation with the China commercial trade policy affected the principal indexes of the global economy.

In this article we will cover how can the tweets be analyzed by the computer in order to indicate a sentiment output between three options, *negative*, *positive*, *neutral*. Everything will be explained in detail using diagrams and code snippets to help with concepts understanding.
After performing a deep research we can figure out that deep learning techniques outputs state-of-art performance at text processing tasks and most researchers are currently studying this matter.
From the available deep learning models, the convolutional neural network (CNN) is the best fit for our purpose which is text classification.

We will see each step involved in the text classification, from how tweets are inputted into the CNN to how does the CNN outputs a accurate prediction.

1 Introduction

This article represents a study of how can we develop an algorithm that analyze the sentiment of a text using machine learning principles. In short, we want a model to which we input a phrase and it outputs the sentiment between three possibilities, negative, neutral or positive.

We are going to make a straight forward approach focusing directly the problem definition and explaining how it can be solved, thus, the structure of this document is directly related to the questions asked in the test sent by the company. This is something that makes this paper useful since it is thought to be a *how to* guide when creating such a project.

1.1 Motivation

I had the luck of studying Deep Learning courses at California State University San Marcos. There we developed a stock price prediction algorithm helping ourselves form an open source project form GitHub in order to conduct a relevant study and apply the concepts learned during the semester.

This paper involves the creation of a model that instead of predicting a number, outputs one out of the three possible sentiments. This means that I will need to apply concepts of Natural Language Processing apart form the deep learning and AI knowledge.

Doing this work I will enhance my machine learning knowledge, I will learn text processing for training purposes and hopefully it will be useful for the community of researchers interested in this area.

1.2 Problem definition

We assume that we have a data-set of 200 millions tweets, labeled with *"Neutral"*, *"Possitive"* or *"Neggative"* sentiments organized in 200 json files, each one containing around 200 thousands tweets.
We want to create a model that is able to receive a sentence as input and output one of those sentiments.

The questions purposed by the company, which I used as a guide when creating this document, are:

- What steps should we follow to create the desired model?

- What question should we ask about the data-set and how should we manipulate it?
- What model architecture do we choose and how do we create it?
- What metrics do we choose and what plots do we print in order to evaluate the model?
- How do we use and support the model while in production?
- Assuming the model is successful but we are asked to create something new. What other models could we produce with this data-set?

2 Project management, Technology and Tasks

The questions described in *section 1.2* gives us an idea of how should we proceed in order to construct the algorithm, however, in order to successfully create the project we need to know beforehand what machine learning technology are we going to use and clearly specify the different parts involved, dividing the project in a series of defined steps that we should treat and solve individually.
The entire project will consist of joining those steps together, making them work harmonically in order to solve the problem we face.

2.1 Selecting technology for the model architecture

Researching about the matter we find that there are many studies about sentiment classification using machine learning techniques.
As we can see in [6] and [25] most of the work done in text sentiment analysis revolves around extensive feature engineering which is both, labor intensive and likely to be too domain specific. Those approaches use machine learning models such as *Naive Bayes (NB)*, *Support Vector Machine (SVM)*, *Stochastic Gradient Descent (SGD)* and *Maximum Entropy (ME)*. The features that these models use in order to perform the classification are word and character n-grams (number of words with all character in uppercase, number of hashtags, number of exclamation signs, number of negated contexts, etc), lexicons, emoticons and some hand crafted features.
We can see in [6] that the accuracy of these models varies between 70% - 90% depending on the features, the number of sentiments and the model we chose. However, the success of these machine learning classifiers depends

on how well the features are defined, involving a large amount of work from domain experts. Therefore, we will approach the problem in a different way, using deep learning to construct our model.

Deep learning models have the advantage of automatically capturing arbitrary patterns like features, providing a meta-level feature representation that generalizes well on new domains. Apart from not having to specify the features, this means that we could train the model created for tweets sentiments analysis to analyze the sentiment of movies reviews, or any other business implementation that is of our interest.
Deep learning is a subset of machine learning that introduces the artificial neural networks. There are two kind of neural networks best suited for NLP tasks, *convolutional neural networks (CNNs)* and *recurrent neural networks (RNNs)*. As stated in [5], CNNs are best suited at extracting local and position invariant features (angry terms, sadness, abuses, named entities) while RNNs works better when classification is determined by a long range semantic dependency (language modeling, machine translation). Furthermore, CNNs are much faster that RNNs, around five times faster.

Considering the specifications of our problem, we want to classify tweets between *"negative"*, *"neutral"* or *"positive"*. It is important that our model is able to detect features, like angry terms, whatever position they have in the phrase or tweet. What's more, thousand of tweets are posted every second, specially if some relevant event that makes the markets shake just happened, therefore, since we want to give our customers the information in real time, being fast is another important factor for our model.

As a result of the above description, we will implement the machine learning model using a convolutional neural network (CNN), because of its position-invariant features detection and quickness.

2.2 Steps to follow at creating the project

Once we decided the machine learning technology on which we will develop our solution, now it's time to plan the development of the project. Since we are going to analyze text using a CNN, the project can be divided in two sections, text processing and model creation. Each one of them has several steps to implement.

1. Text Processing Steps
    - Classify the data-set
    - Balance the data-set
    - Pre-process the tweets
    - Decide word embeddings parameters
    - Generate vocabulary and word embeddings
    - Split between training and testing set
2. Model Creation Steps
    - Define convolution layer and kernels
    - Define max-pooling layers
    - Choose hyperparameters (activation function, loss function, optimization algorithm, hidden layers, learning rate, etc)
    - Create fully connected layer
    - Train the model
    - Test the model

3 Data-set Manipulation

The first thing we need to check is the distribution of the classes (sentiments labels) in the data-set. This means we need to check that there are relatively an equal amount of tweets labeled as negative, positive and neutral. When there is an imbalance in the data-set, the model is overwhelmed by the large classes at training, ignoring the small ones. As a result, our model will be simplistic and inaccurate, favoring the predominant class when analyzing the sentiment of phrases.
We also need to test our model before production, then we need to use a part of the data-set for training purposes and a part of it for testing purposes.

Our data-set is immense, with 200 million of tweets, then we do not have to worry about a lack of training examples. This allows us to perform a simple strategy as unsampling to balance the data-set. Unsampling means that we throw away unwanted samples in order to achieve the required balance. This could be reached with a simple script where we load all the tweets in the data-set and we sort them between positive, negative and neutral. We see which is smallest amount and we throw away samples from the other classes in order to achieve a balance.
Of course we need to check that there is a significant amount of tweets labeled with each sentiment, this strategy would give us a problem if only 5% of tweets are labeled as neutral for example, since in that case we will discard most of the data from the data-set. However, with a data set of 200 millions this usually does not happen, and even in the case that two of the classes represents about 20% of the data each, so we throw away 40% of the data, we would still have 40 millions of tweets by each class, more than enough to train and test the model.

Classifying tweets is a easy task for humans, however, the complicated sentence structure, sarcasm, figurative language and other tweets characteristics make this task really complicated for computers.

3.1 Text pre-processing

We need to clean the tweets in order to make them easy to process by the computer and to avoid the occurrence of problems like overfitting. This is the text pre-processing phase and the methods we should perform are:

- **Remove URL's**: They do not deliver any predictive power since we cannot extract any sentiment information. This can be achieved using a regular expression.

  ```
  text = re.sub(r'http.?://[^\s]+[\s]?', '', text)
  ```

- **Remove accounts mentions**: If an airline has been associated with poor comments in the data-set, our model could use that feature to make predictions, associating that name with the negative sentiment. This can be achieved using a regular expression.

  ```
  text = re.sub(r'@\w+', '', text)
  ```

- **Remove Symbols and Digits**: Points, Commas, Hashtags, Numbers and all other symbols should be removed. This can be achieved using a regular expression.

```
text = re.sub('[^a-zA-Z\s]', '', text)
```

- **Emojis/Smileys**: They are represented with codes or punctuation. We can use of the python library *emoji* to convert each emoji code to a label, while for smileys ":-)" we need to create our own list, or use a open source GitHub one.

```
#Part for smileys

#List of smiley and their conversion.
#{"<3" : "love", ":-)" : "smile", etc...}
words = text.split()
reformed = [SMILEY[word] if word in SMILEY else word stays same]
text = " ".join(reformed)

#Part for emojis
text = emoji.demojize(text)
```

- **Contractions/Slang cleaning**: We need to convert contractions like *you've* to *you have* and slang like *r u* to *are you*. We should look for an open source library or create our own list for this purpose.

```
#We have to create list of contractions and slang and their conversion.
#{ "you've":"you have", "luv":"love", etc...}
text = text.replace("’","'")
words = text.split()
reformed = [List[word] if word in List else word stays same]
text = " ".join(reformed)
```

- **Fix Misspelled Words**: People make typos like *cudtomers* instead of *customers*. We could use libraries to detect and fix misspellings, however, they are slow and this is not acceptable in production. To achieve a better result we will need to write a spelling corrector using regular expressions as you can see in [18].

```
text = re.sub(r'\bcudtomers\b', 'customers', text)
```

```
text = re.sub(r'\bppl\b', 'people', text)
text = re.sub(r'\basap\b', 'as soon as possible', text)
text = re.sub(r'\biphone\b', 'phone', text)
```

- **Remove Stop Words**: Those are words that a search engine has been programmed to ignore since they do not provide value to machines understanding of a text. We could use the Python library *NLTK* for this purpose, however, its list of stop words include words that potentially convey negative sentiments like not, don't, hasn't, therefore, we need to either edit the list to exclude those words or not remove the stop words at all.

- **Convert to Lower Case**: To avoid case sensitive issues we convert everything to lowercase. We employ the Python function *.lower()* for this.

```
text = text.lower()
```

- **Remove White Spaces**: After applying all the previous steps, text could end up with extra white spaces that we should remove. We could achieve this with a combination of a regular expression and the functions *.lstrip()* and *.rstrip()*

```
text = re.sub("\s+", '', text)
text = text.lstrip()
text = text.rstrip()
```

- **Stemming**: Stem is the part of the word to which you add inflectional affixes such as -ed, -ize, -s, etc. Stemming is the process of removing these inflections in words, so it may result in words that are not a valid word in the language. We can use the Python library *NLTK* for this purpose, as you can see in [8].

- **Lemmatization**: Unlike stemming, it reduces the inflected words properly ensuring that the root word belongs to the language. A lemma, is the canonical form or dictionary form of a set of words. For example, runs, running, ran are all forms of the word run, therefore, run is the lemma of all these words. We can use the Python library *NLTK* for this purpose, as you can see in [8].

- **Train / Test splitting**: To ensure our model performs well before production we have to test it. This is achieved by using some tweets from the data-set to see if the sentiments the model analyze are correct. We can use 70% of the data for training purposes and 30% for testing.

3.2 Tokenization and Embeddings

Once we have all the text pre-processed we need to covert it in a form that the algorithm could understand and work with. As stated in [4], neural networks can only learn to find patterns in numerical data, then before feeding it we have to convert each word into a numerical value. This called word encoding or *tokenization*.
Tokenization consist of encoding each word in the data set to a unique integer value, called a token. Usually, the values are assigned based on the occurrence of the word in the data-set. The word that appears most frequently will have the associated token *0*, the next more frequent word the token *1*, and so on. This word-token association is represented in a dictionary that maps each unique word to their token value, as we can see in "figure 1 ".

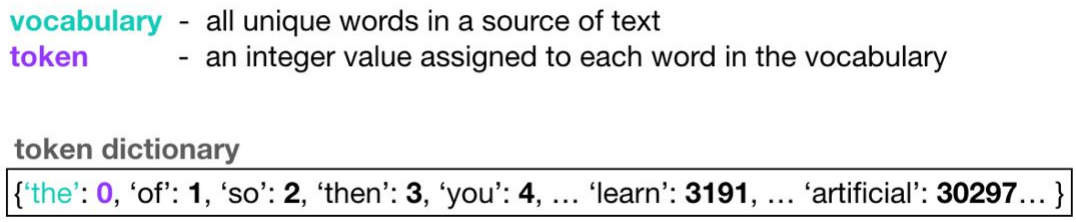


Figure 1: Word-Token Dictionary

We treat each of the tweets as a list of words in a sequence and then we use the token dictionary to convert this list into a vector of integer values. "Figure 2 "shows an example of a sample text and its tokenized vector.

These token values do not have much conventional meaning, our algorithm could think of the value 1 being closer to 2 and farther from 1000, it could think of the value 10 as an average of 2, however, there is no logical relation between the tokenized words.
Then, we need to perform another encoding step to get rid of the numerical

Figure 2: Tokenized phrase

order of the tokens or a step that represents the relationship between words.

There are several ways of performing this step, however the most common one is called word embedding.
Word Embeddings, see "figure 3 ", are vectors of a specified length represented as one word. The values in each column represent the features of a word rather than any other one.
To achieve this we will use the *Word2Vec* model which is implemented in Python by the library *Gensim*. As stated in [7] Word2Vec is an algorithm, combination of two techniques, *Continuous Bag of Words (CBOW)* and *Skip-gram* model. Both of these are shallow neural networks that learn weights which act as word vector representation. *CBOW* tends to predict the probability of a word given a context while *Skip-Gram* predicts the context of a given word.
As a result, words that show up in similar contexts such as *apple*, *pear*, *mango*, will tend to have similar vectors, they point out in a relatively similar direction. These word embeddings have some really nice properties like allowing us to easily find similar words or semantic relationships between words.

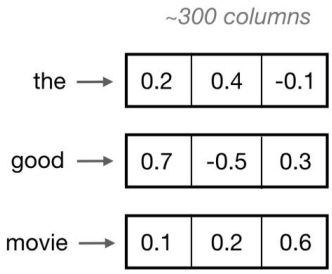


Figure 3: Word Embeddings Vectors

The word embeddings generated with the Word2Vec algorithm are ready

to be used as inputs to our Convolutional Neural Network first layer. Using *Gensim* library we generate a dictionary of word embeddings, which will store all the embedded vectors for each word in our data-set.
Then, we can generate the input matrix to our CNN by getting the embedded vector for each word in the phrase we want to input.

4 Model Architecture

Even if we are going to cover several deep learning parameters, like of what is a neuron, weight factors, input and output layer, hidden layers, bias unit, activation function, backward propagation, loss function, optimization algorithm and other neural networks terms, the reader needs to have a basic understanding of them to understand all the concepts in this section. We'll start this section explaining directly the Convolutional Neural Networks CNNs. Please visit [26] to gain knowledge about artificial neural networks.

4.1 Convolutional Neural Networks

CNNs were first introduced because of image processing in order to deal with Computer Vision tasks, with the idea of enabling machines to view the world as humans do. A Convolutional Neural Network is a deep learning neural network that that can take as input an image, assign importance (weights and biases) to various aspects/objects in the image and be able to differentiate between them. "Figure 4 "gives us a graphical representation of a CNN. The architecture of these neural networks was inspired by the organization of the visual cortex in the human brain. Individual neurons respond to stimuli only in a restricted region of the visual field, known as the *receptive field*, meanwhile a collection of such fields overlap to cover the entire visual area.

An image in it simplest form is a multidimensional matrix of pixel values. A CNN is able to successfully capture spatial and temporal dependencies through the application of relevant filters. The original role of the CNN is to reduce the images into a form which is easier to process without loosing features.

The name Convolutional derives from the fact that int the first hidden layers of these neural networks we apply convolution and pooling functions as activation functions, both concepts borrowed from the field of computer vision. The usual architecture of a CNN consists of convolutional layer,

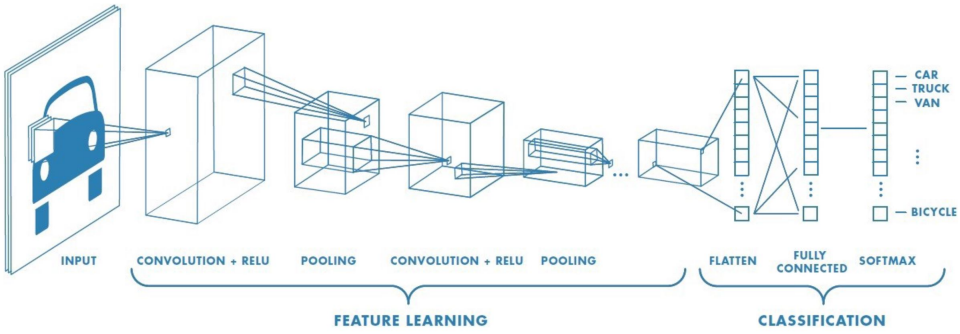
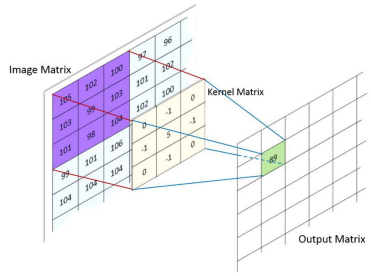


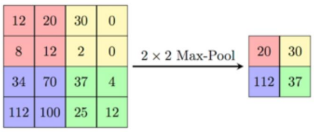
Figure 4: Graphical Representation of a Image Processing CNN

pooling layer and fully connected layer.
Convolution operates on two signals (in 1 dimension) or in two images (in 2 dimensions). We could think of the first signal as the input and the second signal, called the kernel, as the filter. Convolution takes the input signal and multiplies it with the kernel, outputting a third modified signal. The objective of this operation is to extract high-level features from the input. There are two types of results. Where the output is reduced in dimension (this is called Valid Padding) and where the output has the same dimension or even bigger (this is called Same Padding).“Figure 5 a) ”shows a graphical representation of convolution as an image processing example. The *kernel* is represented as a smaller matrix sliding over the entire input image, changing the value of each pixel in the process, outputting a third image with a smaller dimension, thus we are applying Valid Padding.
Pooling is a sample-based discretization process where we down-sample an input reducing its dimensions and allowing for assumptions to be made about features contained in the selected sub-regions. This is to decrease the computational power required to process data and to extract dominant features. There are two main types of pooling, max pooling of which we can see a graphical representation in “figure 5 b) ”and average pooling. As their name suggest, one of them is based on picking the maximum value form the selected region while the other is based on calculating the average of all the values.

Convolution layer and pooling layer are highly related together in the usual architecture of a CNN, this is, we will always have a pooling layer after a convolution layer, thus, they can be considered as a single layer of



(a) Convolution graphical representation



(b) Max pooling graphical representation

Figure 5: Convolution and Pooling illustration

the CNN.
Depending on the complexity of the input, the number of these layers can be increased for capturing low-level details at the cost of more computational power. After the above process, we have enabled the model to understand the features.
If we are performing 2 dimensional convolution, this is, the kernel matrix moves in two dimensions over the input matrix, the result of the latest convolutional-pooling layer will usually be a multidimensional matrix. We need to flatten this final output into a vector containing one value per index, to be able to feed it to a regular neural network for classification purposes.

Flattening involves transforming the entire pooled feature map matrix into a single column which is then feed to the neural network for processing. After the flattening step we end up with a long vector of input data as seen in “figure 6 ”. We do this because we need to insert this data into an usual fed forward artificial neural network.

We call the fed forward neural network to which we input the flattened vector the fully connected layer. The features filtered in the convolution-pooling steps are encoded in the flattened vector, they are already sufficient for a fair degree of accuracy in recognizing classes. The role of the fed forward neural network is to take this data and combine features in order to make a prediction about the probability of each class.
We are assuming the reader has a good knowledge about artificial neural networks, therefore I am not going to cover them from scratch, however, I will summarize the concepts involved in the fully connected layer that we

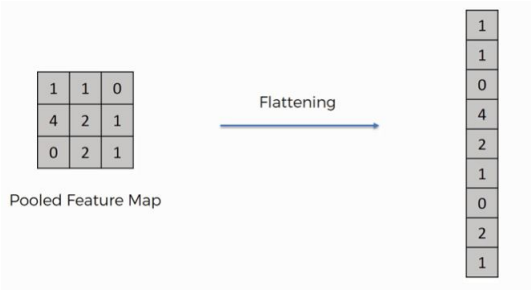


Figure 6: Graphical representation of flattening

could implement. For a better understanding of the concepts, check [26]

The Artificial Neural Network (ANN) we will use as fully connected layer for our CNN has as input layer the flattened vector, each vector value is going to feed every neuron on the first hidden layer. The neurons in the hidden layer will be triggered by a Rectified Linear Unit function *(ReLU)*. Mathematically it can be described as $f(x) = max(0, x)$, you can see a graphic representation in "figure 7 ". This is the most commonly used activation function. It has the advantage that the output is not enclosed into the interval $[-1, 1]$ like *Tanh* or the interval $[0, 1]$ like the *Sigmoid* activation functions.
The output layer will consist of neurons triggered by the *Softmax* activation function. In "figure 8 "we can see a diagram with all the components of our Convolutional Neural Network, the same way we can have various convolution and pooling layers, the fully connected layer can have several hidden layers, even further, we can mix the activation functions between hidden layer. we could have neurons with *ReLU* activation function in the first hidden layer while having neurons with the *Sigmoid* or *Tanh* activation functions in the successive hidden layers. However, researches suggested that the ReLU activation function works the best for classification purposes, being this another reason it is the most commonly used.

"Figure 9 "shows us a neuron where X_i are the inputs, W_i are the weights associated with every input, b is the bias unit. This values, the weights and the biases, are randomly assigned by the algorithm and automatically updated through the trained process. Ass we can see, the neuron just realize the sum of the inputs multiplied by the weights, $(\sum_{i=1}^{n} X_i * W_i) + b$ and then adds bias term. The result is processed by the activation function which

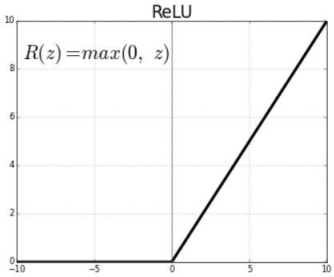


Figure 7: Graphical representation of the ReLU activation function

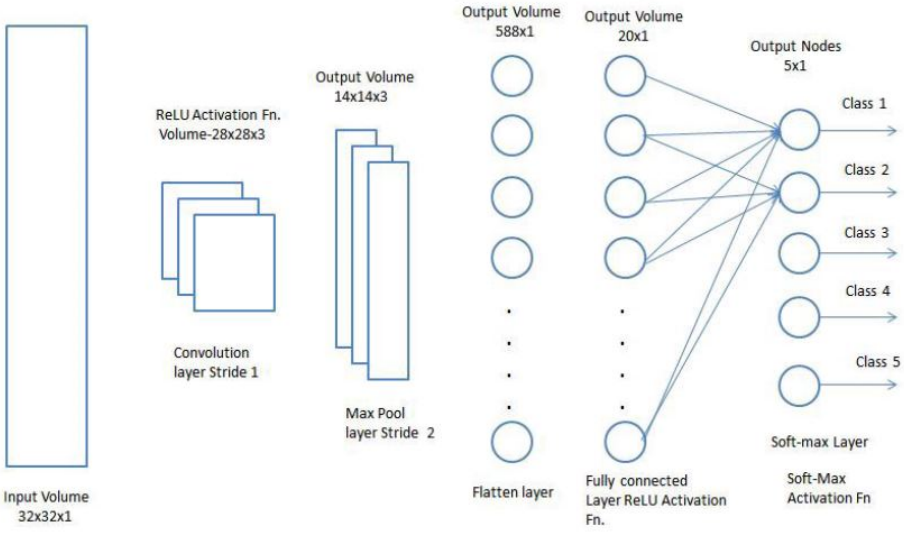


Figure 8: Diagram of the described CNN

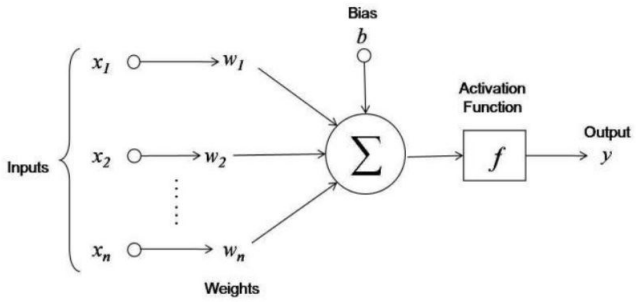


Figure 9: Diagram of an Artificial Neuron

returns the output y. The role of the activation function is to encode the result of the summation between a fixed scale which is the same in every neuron in the same hidden layer, so the neural network can learn from the different outputs form the different neurons. This is the reason we cannot mix neurons with different activation function on the same layer, we need the output of each neuron in the same layer to be scaled between the same range, so relations between outputs can be computed. This allows for weight and biases recalculation when training. This is what makes our model have a better prediction each time.

As the "figure 10 "illustrates, the training is an iterative process for all known labeled input examples. Each time an input advance through the neural network, we compute the error between the prediction and the expected result, this is done through the **loss function**. Once we know the error, we adjust the parameters values, weights and biases, in a way that after each iteration, the value of the error obtained is getting smaller. This is performed through the **optimization algorithm**. Once we have done this process with all the labeled samples (the training set), we completed what is known as an **epoch**. Performing n epochs means repeating the before described iterative process n times with the same data-set. This could lead to **overfitting**, meaning that the definitive weight and biases parameters obtained after the training are over adjusted to the features inside our data set. Our model will work perfectly with samples inside the data-set, because we repeated and adjusted the parameters several times using the same examples, however, when in production and analyzing inputs outside the data-set, the model could give us bad predictions because the relations between all the samples in the data-set are not the same as with this sample

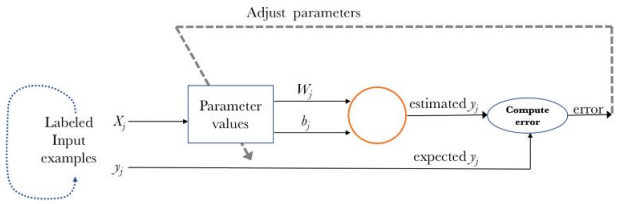


Figure 10: Training Iteration

coming from outside.
One important factor to avoid this overfitting is having a big amount of samples in the data-set representing all the possible classes classification, so when an input from outside the data-set is introduced, this input has similar features with many other that are inside the data-set, therefore our model will perform a good prediction. This is the reason that creating a global free accessible data-set is important for some companies. For example, autonomous cars will perform better at detecting objects, animals and predicting situation the more samples of this situations they are trained with. If all car companies publishes their training images in the same data-set so is worldwide or nationwide accessible, one autonomous car will be able to learn how to perform even in situations it has never faced, just because some other company has faced that situation and uploaded it to the data-set to perform the training.

As mentioned before, in our case we have an immense data-set with 200 million tweets, which in the best case is around 66 million samples per each class (we have three classes *negative*, *positive* and *neutral*) and in the worst case, assuming we only keep 60% of the data-set, 20% for each class, we have 40 million tweets per class. This will allow us to perform many epochs and still expect our model to perform well when in production, then not facing overfitting.

The first step of the training process is the **forward propagation**, when the inputs cross through the hidden layers towards the output layer. We use the loss function to calculate the error between the predicted output and the known correct result. Once the loss has been calculated, we use this information and we propagate it backwards starting from the output

layer and propagating the loss information to all neurons that contributed directly to the output through the hidden layers. Hence, the name of this process is called **backward propagation**. Once we have spread the loss to each hidden layer, we use this parameter to modify the weights and bias in such a way that the error is getting smaller with each iteration. These parameters are modified using a function called **optimization function**. The algorithm that makes effective the parameters updating process through the optimization function is called the **optimization algorithm**.

Optimization functions usually calculates the gradient, the partial derivative of the loss function with respect to the weights, and the weights are modified in the opposite direction of the calculated gradient. As we know, the gradient vector at a point (x, y) points towards the direction of the greatest rate of increase of the function at that point. We want to minimize the loss function because it represents the error between the predicted output and the correct known output, the smaller the loss function the smaller the error, therefore the prediction is better. With each iteration we move towards the minimum point of the loss function a specific quantity, which is called the **learning rate** and represented as α.
In function of the outputs we are looking for, we have three main categories of loss functions to choose. Regressive loss functions, used in the cases where the target variable is continuous. Embedding loss functions, when we deal with problems where we have to measure if two inputs are similar or not. Classification loss functions, which is our case, are used when used when output variables are probabilities. This probability is called the score of the input, which represent the confidence of the prediction. We have three sentiments, therefore we will have three outputs and each output will represent the probability of its correspondent class.

Optimization algorithms fall in two classes. Constant learning rate algorithms and adaptive learning rate algorithms. As the name suggest, the first class uses a constant value for the learning rate parameter (α) while the second modifies it during the training process. Researches suggested that adaptive optimization algorithm performs better, and from these, the Adaptive Moment Estimation ($ADAM$) is the most popular. It works with momentums of first and second order. It stores an exponentially decaying average of both, past gradients and past squared gradients. This optimization algorithm compares favorably to any other adaptive learning rate algorithms. Thus, is the one we should implement in our artificial neural network.

Finally, we will cover the **Softmax** activation function, used for classifying between two or more classes. The softmax regression is a form of logistic regression that normalizes an input value into a vector of values that follows a probability distribution whose total sums up to one. This characteristic allows us to accommodate as many classes as we want in the output layer. The mathematical representation of the softmax is $f(x) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$ where z is a vector of the inputs to the softmax layer, j indexes the amount of inputs then $j = 1, 2, 3, 4.....$ and i indexes the number of outputs then in our case $i = 1, 2, 3$ since we have three classes.

Reached this point, we have seen the complete architecture of a convolutional neural network. We have explained everything from convolution and pooling layers to the output layer where we get the probabilities of each class.
As final comment, is important to say that finding the correct value for the hyperparameters, configuration variables that are external to the model and whose value cannot be estimated form the data (number of hidden layers, number of neurons per layer, activation functions, loss functions, optimization function, optimization algorithm, learning rate, number of epochs, etc), involves practicing and testing the model using different combinations in order to see which one outputs the best result.

4.2 CNN for text processing

As far we deeply covered how a Convolutional Neural Network is formed and how it works. Now its time to focus on how does the matrices are formed and how convolution and pooling are performed in order to analyze text sentiment.
We saw that the convolution operation in 2D consist of applying a kernel to the input matrix, seen as a matrix sliding through the input matrix. Each position in the kernel matrix have a specified weight associated. The kernel weights are multiplied by the values of the input matrix and then the results are summed out, together with a bias unit, to get an output.
In the case of text classification, the role of the kernels is to look at embeddings for several words in the tweet. Thus, as we can see in the “figure 11 ”the width of the kernel will depend on the length of the embedding vectors we have created for each word, while the height of the kernel can variate and it represents the number of words in the tweet we are looking at in the same time moment.

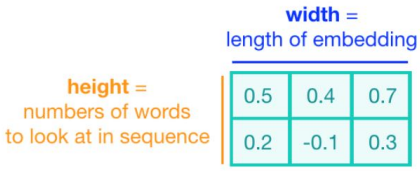


Figure 11: Height and Width of a Kernel

A CNN will include many of these kernels whose weights and biases will be updated during the training process. The kernels are designed to look at a word and its surroundings to output a value that captures some relations between them. We recall that the embeddings generated for with the Word2Vec algorithm are similar if the word are also similar, therefore, the outputs generated by the convolution operation are going to be similar when looking at a similar pair of words. We can see in "figure 12 "how the output value for the words *good movie* is similar with the output value for the words *fantastic song*. This happens because convolution is a linear operation.

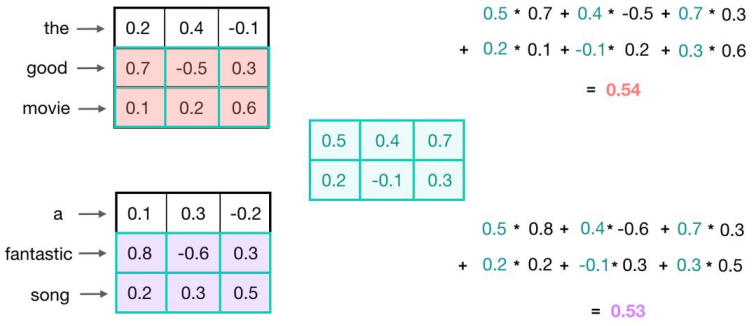


Figure 12: Convolution outputs similar results when looking at similar pair of words

To process a entire phrase or tweet, the kernels will have to slide down through all the words that forms it. This is called one dimension convolution (**1D**) because the kernel is moving only in one dimension, sliding down one by one through the list of words embeddings, see "figure 13 ". One

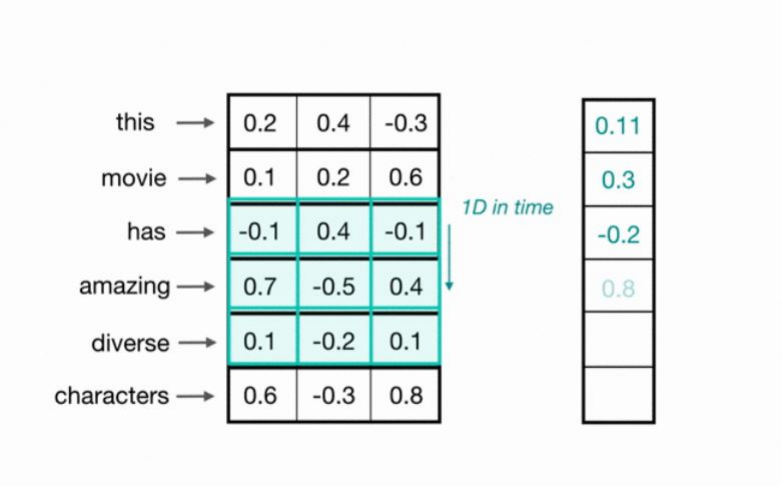


Figure 13: 1 dimension convolution

convolutional kernel is not enough to detect all the needed features for the classification task. To build a CNN able of learning a variety of different relationships between word we have to use many kernels of different heights. One convolution layer contains several kernels, in theory we could have as many kernels as we want per layer. In [12], Yoon Kim et al. used 300 kernels, 100 for each height he chose, 3, 4 and 5. They didn't set a bigger value than 5 because words that are further away are less relevant with respect to identifying patterns in a phrase. For example, in the phrase *what a bad movie, however, I enjoyed the theater* word that are close together like *a bad movie* are more related than further apart words like *movie enjoyed theater*.

Once we've seen how a convolutional operation produces a feature vector representing features in a sequence of words embeddings, we now want to identify the high-level features, those who are most important. To achieve this we apply the pooling operation.
As we see in "figure 14 ", using a max-pooling operation forces the network to retain only the maximum value from each feature vector, which should be the most useful local feature from all the set of words the kernel has processed into that vector. The max-values outputted by the last pooling layer are already individual vales ready to be inputed to the fully connected layer described in *section 4.1*.

The length of a tweet may vary, then the number of words embeddeds

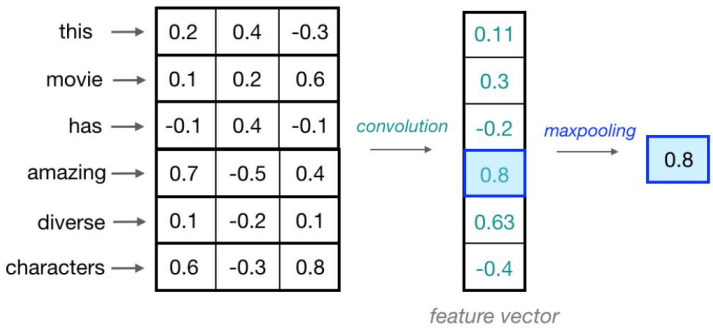


Figure 14: Max Pooling over a feature vector

that we have as an input may change. We cannot dynamically change kernels height to generate same amount of outputs, then, we have to apply zero-padding in order to ensure we have the same amount of outputs from the pooling layer, this is, each tweet we input has to be processed in a way to give the same amount of inputs to the fully connected layer.

Zero-padding consist on considering a maximum height for the input matrix and if the tweet is shorter than the maximum height we add zeros in the empty cells. Recall that the input matrix has always the same width, which is the size we chose for the embedded vectors, and the height is determined by the amount of words in the tweet, which can vary with every tweet we take.

Zero-padding technique can easily be applied in our case because tweets have a limited number of characters, so we can surely estimate a maximum amount of words which will be imputed to our model and fill the empty cells with zeros.

Convolution is a linear operation, then the outputted feature vector will be just a vector full of zeros if we are looking only at empty cells. Since the vector will only consist of zeros, the value outputted by the pooling operation and imputed to the neurons of the fully connected layer is going to be a zero. The term corresponding to the zero input in the sum that the neuron realizes will be zero, because zero multiplied by any weight w value is zero.

Therefore, the empty words in the input matrix will have no weight over the classification decision.

5 Model Implementation

Now we fully understand both aspects of the model we want to create, we know how Convolutional Neural Networks operates, we know how are them used to perform text sentiment analysis and we know what text preprocessing operations we need to do. “Figure 15 ”is a graphical representation of all the steps involved in the sentiment analysis using a CNN. The diagram represents a two classes sentiment classification, keep in mind that for classifying between more classes the only thing we would have to do is to add one more neuron with the softmax activation function in the output layer.
We know what every parameter in the model represents, then we are ready for implementation, to develop an algorithm that introduces the text preprocessing steps and the model creation, training, testing and classification.

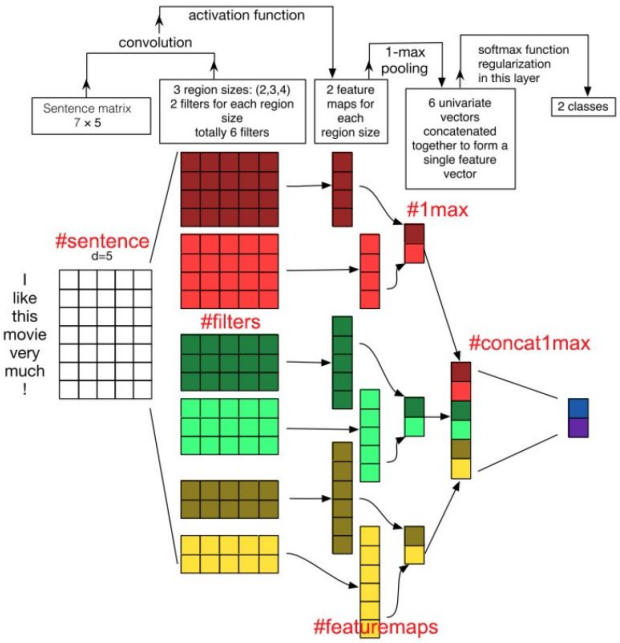


Figure 15: Graphical representation of the CNN text classification process

The focus of this paper is not to develop the correct implementation of the project. Instead it is thought to be a how to do it guide, therefore in

the following sections we are going to see explanations and code snippets of how to perform the implementation.
The code snippets that I am going to present are extracted from an open source project available in GitHub [29] which I recommend visiting.

5.1 Data-set pre-processing

Our data-set consists of 200 json files. Each one of the files stores the information in a similar way that we see on the code below:

```
{ "JSONFile":
    {"person1":
      { "tweet": "this is a sample tweet posted by person1",
        "sentiment": "positive"
      }
    },
    {"person2":
      { "tweet": "this is a sample tweet posted by person2",
        "sentiment": "negative",
      }
    },
    .
    .
    .
    .
}
```

There are several libraries in Python to work with JavaScript Object Notation *json* files, one of which is the *JSON* library. These libraries have many functions implemented to process the json files.
The first thing we want to do is filtering all the tweets in the data-set in order to see the balance between sentiments, recall that we want a similar amount of tweets for each sentiment for training purposes in order to avoid our model favoring one of the sentiments when classifying. Since we have around 200 millions tweets, reading and storing them in the static memory may result in computation problems. If that is the case, the strategy we could use is reading every tweet from each json file and store the data-set in three new files, one per sentiment, where we are going to input all the tweets. We just need to keep track of how many tweets are classified as positive, negative or positive. This can be easily done by storing this information in

three separate variables.

Once we know how many tweets of each class we have, we need to develop a logic function that calculates how many tweet of each class we will throw away. This function could return another three variables indicating how many tweets of each sentiment we have to throw away.
After we have balanced the data-set, we now have to apply all the text pre-processing steps indicated in *section 3*. Then we remove the URL's, hasthtags, symbols and digits through regular expressions as shown before. We process the emojis, smileys, slangs, misspelled words, contractions, stop words, stemming and lemmatization through python libraries or creating our own lists. After this we convert everything into lowercase and remove the white spaces. Code snippets with this procedures can be seen in the correspondent *section 3.1.*

After this we need to generate the words embedding for all the words in the vocabulary, which are all the words in the data-set. As suggested before, Word2Vec is an algorithm created for this purpose which generates embedding vectors that are similar for words that are used in the same context. This algorithm is implemented by the python free library *Gensim*. Gensim provides the *Word2Vec* class to work with this model. Learning a word embedding involves loading the tweets and provide them to the constructor of a *Word2Vec* instance. There are several parameters allowing us to configure the process, below I describe the most useful:

- **size**: this parameter determinate the length of the embedded vector for each word
- **window**: similar to the width of the kernel, this parameter symbolizes how many words around the target word are we looking at
- **min_count**: this is the amount of time a word needs to appear in the tweets in order to consider it when training, words with occurrence less than this mincount will be ignored
- **sg**: this parameters indicates the training algorithm, either CBOW (0) or Skip Gram (1)

The code below, from [3], is an example of how to work with a Word2Vec model to generate words embeddings for a small list of sentences. I've added comments to explain the meaning of each command line. For a very large

data-set as ours we need to generate a iterator that progressively loads tweets and inputs them in the Word2Vec model.

```
# Load the Word2Vec model
from gensim.models import Word2Vec

# Training data
sentences = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
['this', 'is', 'the', 'second', 'sentence'],
['yet', 'another', 'sentence'],
['one', 'more', 'sentence'],
['and', 'the', 'final', 'sentence']]

# To train the model we just need to pass the sentences to the constructor
# together with all the required parameters.
model = Word2Vec(sentences, min_count=1, size=5)

# This command prints basic information about the model
print(model)

# This command outputs vocabulary list of words
words = list(model.wv.vocab)
print(words)

# This command will show us the embedded vector for one specific word
print(model['another'])

# We can save the trained model in separate file
model.save('model.bin')

# We use this command to load an already trained model
new_model = Word2Vec.load('model.bin')
print(new_model)
```

As mentioned in the code above, the trained model consist of embedding vectors each one of them representing one word form our data-set. All of those vectors will have the same length, indicated by the parameter *size* and we can access the embedding vector of one word using the command *model['word']*.

Therefore, assuming we want to input a tweet in our sentiment analysis CNN, we just need to get the embedding vector for each word in the tweet and form the input matrix for our CNN. As mentioned before, we need to perform zero padding in order to ensure the input matrix has the same size for each tweet. This can be achieved by selecting a maximum tweet length.

There is one more important factor to mention in this section. This is, what if one word form a tweet which we input is not in the model generated by the Word2Vec algorithm ?. This can happen in production when analyzing tweets that are from outside the data-set.
Gensim allows us updating an already trained model with new tweets by using the *buid_vocab()* and *train()* functions just as we can see on the code snippet below.

```
# First we need to load the model
new_model = Word2Vec.load('model.bin')

# Now we can upload new word in the model
new_model.build_vocab(new_tweets, update=True)
new_model.train(new_tweets)
```

5.2 Model creation

So far we have seen everything about the text pre-processing, from the unstructured data-set to the embeddings vectors matrix which we input to our CNN. Then, the logical next step is to see how can we implement the Convolutional Neural Network for text processing.
There are several deep learning libraries for pyhton. For the code snippets bellow we assume we are using the free open source TensorFlow library, originally developed by the Google Brain team in 2017.

First of all we need to decide the hyperparameters of our model. We know that as input we have a fixed size matrix of width the size of the embeddeds vector and height the considered maximum length of a tweet. We also have seen that for text classification the width of the kernel has to be the same as the width of the embedded vector, then the output of the pooling layer is just one value per kernel, this is, we will only have one convolution and pooling layer in our CNN.
Since we will have individual values outputted by the pooling layer, they are ready to be inputted to the fully connected layer. The output layer needs

to have softmax activation function with three neurons.

The code snippet below represents the implementation of the convolution and pooling layer using the Tensor Flow library. In order to learn several relevant features from the inputted tweet, we need to define several kernels or filters of different sizes. For every filter size we need to calculate the filter matrix W and the bias term b. We perform the convolution using the *tf.nn.conv2d(input, filters, strides, padding, data_format='NHWC', dilations=None, name=None)* function. Before pooling, we can apply the activation function ReLU to get rid of the negative values (recall that ReLU returns the same value if it is zero or bigger, and returns zero for values smaller than zero).
After applying the ReLU we get the features vector h, representing all the features that this respective kernel has extracted from the sentence. We then apply the pooling operation using the *tf.nn.max_pool(input, ksize, strides, padding, data_format=None, name=None)* function. This will extract the maximum value of the features vector h and save it in the *pooled* variable. We finally store the pooled feature from this kernel into a list where we store all the pooled features from all the kernels we are applying.
This process is repeated for each kernel we have defined, then we end up having all the pooled features from all the kernels in the list *pooled_outputs*.

```
pooled_outputs = []

for i, filter_size in enumerate(filter_sizes):

    with tf.name_scope("conv-maxpool-%s" % filter_size):

        # Convolution Layer

        filter_shape = [filter_size, embedding_size, 1, num_filters]

        W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")

        b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")

        conv = tf.nn.conv2d(

            self.embedded_chars_expanded,
```

```
        W,
        strides=[1, 1, 1, 1],
        padding="VALID",
        name="conv")

    # Apply nonlinearity
    h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
    # Maxpooling over the outputs
    pooled = tf.nn.max_pool(
        h,
        ksize=[1, sequence_length - filter_size + 1, 1, 1],
        strides=[1, 1, 1, 1],
        padding='VALID',
        name="pool")
    pooled_outputs.append(pooled)
```

The values we obtained in the *pooled_outputs* vector are ready to be inputted to the fully connected layer, however, if we have various kernels of the same size, their pooled features are extracted in one iteration of the previous code. Then, *pooled_outputs* is vector containing several vectors. We need to reshape this vector into a simple flat vector, that contains one feature in each cell.

```
num_filters_total = num_filters * len(filter_sizes)

self.h_pool = tf.concat(3, pooled_outputs)

# Pooled features organized into a flat vector
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])
```

Now we are ready to implement the fully connected layer. As we see in the code snipped below, first we initialize the weights for each input and biases for each neuron. Here, we could have used *tf.layers.dense()* function to create a regular densely-connected neural layer, however, it has been preferred to use the *tf.nn.xw_plus_b()* function which basically does the function of the neuron (recall that the neuron computes the sum of the weighted inputs together with the bias term) and the function *tf.argmax(input, axis=None, output_type=tf.dtypes.int64, name=None)* which returns the index with the largest value across axes of a tensor. Since we input a flat vector, *self.predictions* will be only one value with the position that represents the predicted output.

```
with tf.name_scope("output"):

    W = tf.get_variable(

        "W",

        shape=[num_filters_total, num_classes],

        initializer=tf.contrib.layers.xavier_initializer())

    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")

    l2_loss += tf.nn.l2_loss(W)

    l2_loss += tf.nn.l2_loss(b)

    self.scores = tf.nn.xw_plus_b(self.h_pool_flat, W, b, name="scores")

    self.predictions = tf.argmax(self.scores, 1, name="predictions")
```

Once we have the predicted output, we need to calculate the loss function which we need for training purposes. The code to perform this is straight forward and showed below. In this case we used the mean crossed entropy loss function, however, checking the online documentation we can implement the different loss functions we've talked about in *section 4*.

```
with tf.name_scope("loss"):

losses = tf.nn.softmax_cross_entropy_with_logits(self.scores, self.input_y)

self.loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss
```

Finally, we need to implement the optimization algorithm and train the model. The code below represents how can we implement the Adam optimization algorithm , the most used due to its quickness and reliability.

```
global_step = tf.Variable(0, name="global_step", trainable=False)

optimizer = tf.train.AdamOptimizer(1e-3)

grads_and_vars = optimizer.compute_gradients(cnn.loss)

train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)
```

After training we are ready to test our model and check how does it performs. Of course the code snippets shown in this section are just illustrative to point out how can the theoretical concepts seen in *section 4* be implemented. For a full implementation of the code please check [29].

6 Model Evaluation

Once we have created our model we need to check it works properly before we put it into production.
From the most basic steps like checking reasonable result till more advanced process like checking there is no overfitting and keeping track of the accuracy while in production to detect problems.

Is important to note that the first step into getting a good model is choosing a good combination of hyperparameters, and this cannot be performed without previous training and testing. We have to see what combination of parameters that we need to initialize works better for the classification. To have a starting point we could chose initial values similar to the ones used in other researches.

6.1 Metrics and plots

The most important parameter at which we can look while training is how the loss function is reduced. This should happen each training iteration.
First we need to perform several training processes experimenting with different hyper-parameters. Because of the nature of our problem, we can only change the kernels, hidden layer in the fully connected layer, type of neurons, optimization algorithm, loss function, learning rate, epochs and more parameters related to the fully connected layer.
What we have to do is to test several combinations of this parameters and see which one reduces at maximum the loss function.

After finding a satisfactory combination of parameters, we can employ the testing set to calculate the accuracy of the model. This shows us how many tweets the model has predicted correctly form all the training set, then accuracy is presented as a percentage.
Here is the moment when we have to watch for overfitting. Overfitting refers to the model corresponding too closely to the data-set we have, then it may fail to fit new data coming from the internet, which is exactly what we want the model for.
The validation metrics usually increase until a point where they stagnate or start declining when the model is affected by overfitting. A simple way to check if the model suffers of overfitting is if it performs way better on the training set than on the test set.

As we have seen, it is important to have a balance in the data-set used to train the model. This is, having enough tweets classified with each sentiment. Usually, when something advantageous about some subject happened, tweets published by people about that subject are positive. On the same way, if a disadvantageous event happens, negative tweets will be posted.
With this I want to point out how important is how do we form our data-set. We need to cover all the situations and possibilities when training to have good results when predicting, then, we have to carefully choose for our data-set based on many factors.
For example, if we are going to use the model to analyze sentiment about a specific car maker. We should use in our data-set tweets from when their cars wins competitions (those will probably be positive), tweets from when their cars usually breaks during competitions (probably more negative) and tweets from moments when they perform in a similar way as the average car makers.

6.2 Using the model in production

When in production we use an online API that returns tweets posted by someone in real time about any matter that is of our concern. We usually would use the model to get information about important incidences that are about to have an economic impact, therefore, thousands of tweets per second are going to be posted about that matter if something relevant happens.
Let's consider the new global virus COD-19. Imagine that we use the model to predict people sentiment about global health, following twitter posts from different organizations relevant to the global health. Because of this virus, each one of that organization started posting several tweets a day, when before they maybe posted one tweet per week. Then, the volume of tweets the model has to analyze increased exponentially. We need to be conscious that this happens and have the computational resource to be prepared for it.
In fact, we could track the volume of tweets the model is processing per day and use the increase of that value as a possible indicator of some relevant success going to happen in the world.

Thanks to have chosen the CNNs as the structure of our model, we do not need to specify features text in order to perform the classification. Furthermore, our CNN model learns those features by itself when updating the value of the weights and biases parameters. This allows us to perform text classification for subjects of different domain without performing any mod-

ification to the algorithm.
This is a great advantage, nevertheless, it would not be a good implementation to use only one model to check all the tweets from different expertise domains. A better strategy is to use a copy of the model to analyze tweets form a specific domain knowledge. Otherwise, apart from having a massive data-set which is takes a lot to process, there could be inferences on relations formed between words for different contexts. For example, if we use the phrase *what a fight I just saw!*, in the context of school stories this phrase should be related to negative sentiment, however, in the context of WWC, MMA, and Box reviews this phrase is related with a positive sentiment.
As you could imagine, this kind of misinterpretations can happen between many different contexts and situations, that's why is a good idea to dedicate one model to classify text just within the same domain of expertise.

As time moves on, vocabulary may change together with the way people express themselves. Is is a good idea to take a sample of tweets from the recently published ones for testing purposes. This means constantly keeping track of the accuracy of our model while in production. By doing so, if we detect over time a declining in accuracy (we need to keep some recent tweets and use human interpretation to classify them based on the actual contexts), probably something has changed that makes our model no more valid for the task it used to be.
Then, we have to study what is that change and how can we solve it. Maybe we have to regenerate the data-set and retrain the model, maybe we may need to change model parameters.

7 Other Models

As we have seen during the *section 2*, there are several machine learning options available to perform the task of text classification. Let's say we want to create a new model using the same data-set, for comparison with the one we have just created, even if it is performing well.

Because the not deep learning options involves hand engineered features creations, I would not choose any of them. This involves studying the relations between words and generate complicated rules and algorithm that we would need to develop in order to represent the relations we have studied.

Therefore, the next most interesting model I would generate and study

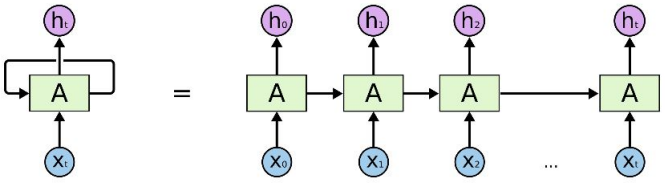


Figure 16: Recurrent Neural Network (RNN)

using the same data-set is a Recursive Neural Network (RNN). These are a class of Artificial Neural Networks where connections between units form a directed graph along a sequence, as you can see in "figure 16 ". From all the inputs (which in our case are the words of a tweet), the neural networks takes the first value X_0 and generates the output h_0. Then, we perform another loop where the neural network takes as input the second value X_1 and also uses the output from the previous step which was h_0 to process the output h_1. Keeping with this sequence, the third value (word in our case) X_2, and the output from the previous step h_1 will be used to calculate h_2, and so on, till we get the final output h_t.

Because of this sequence, the Recurrent Neural Network develops the ability to learn from the previous inputs. We can say them have an internal memory that allows previous inputs to affect the subsequent predictions. This gives RNNs the ability to predict next words in sentences. Even so, they can be used as per text classification task, however, we have preferred a convolutional neural network because of their ability to learn features whatever position they have in the phrase, which works better with classification. Since RNNs keeps track of the input sequence, if we input sentences where the features changes position like *i don't like how the festival was organized, however, I enjoyed my time there* and *I enjoyed the festival, however, i didn't like how they organized it*, then it will require more training for the RNN in order to learn all the features since they perform loops for each word we input, then it will take longer to output the prediction.

8 Bibliography

References

[1] Arunava. *Convolutional Neural Network*. Ed. by Medium. Dec. 25, 2018. URL: https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05.

[2] Denny Britz. *Implementing a CNN for Text Classification in TensorFlow*. Ed. by WILDML. Dec. 11, 2015. URL: http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/.

[3] Jason Brownlee. *How to Develop Word Embeddings in Python with Gensim*. Ed. by MachineLearningMastery. Oct. 6, 2017. URL: https://machinelearningmastery.com/develop-word-embeddings-python-gensim/.

[4] Cezanne Camacho. *CNNs for Text Classification*. Ed. by GitHub. May 27, 2019. URL: https://cezannec.github.io/CNN_Text_Classification/.

[5] Shreya Ghelani. *Text Classification — RNN's or CNN's?* Ed. by Medium. June 2, 2019. URL: https://towardsdatascience.com/text-classification-rnns-or-cnn-s-98c86a0dd361.

[6] Young-Seob Jeong Hannah Kim. "Sentiment Classification Using Convolutional Neural Networks". In: *AppliedSciences* (2019).

[7] Moshe Hazoom. *Word2Vec For Phrases — Learning Embeddings For More Than One Word*. Ed. by Medium. Dec. 22, 2018. URL: https://towardsdatascience.com/word2vec-for-phrases-learning-embeddings-for-more-than-one-word-727b6cf723cf.

[8] Hafsa Jabeen. *Stemming and Lemmatization in Python*. Ed. by DataCamp. Oct. 23, 2018. URL: https://www.datacamp.com/community/tutorials/stemming-lemmatization-python.

[9] Z. Jianqiang, G. Xiaolin, and Z. Xuejun. "Deep Convolution Neural Networks for Twitter Sentiment Analysis". In: *IEEE Access* 6 (2018), pp. 23253–23260. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2776930.

[10] Gautam Karmakar. *Word embedding and Convolutional Neural Network for Sentence Classification*. Ed. by Medium. July 30, 2017. URL: https://medium.com/@gautam.karmakar/word-embedding-and-convolutional-neural-network-for-sentence-classification-89e3f300ce2f.

[11] Joshua Kim. *Understanding how Convolutional Neural Network (CNN) perform text classification with word embeddings*. Ed. by Medium. Dec. 3, 2017. URL: https://towardsdatascience.com/understanding-how-convolutional-neural-network-cnn-perform-text-classification-with-word-d2ee64b9dd0b.

[12] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. DOI: 10.3115/v1/D14-1181. URL: https://www.aclweb.org/anthology/D14-1181.

[13] Charles Malafosse. *FastText sentiment analysis for tweets: A straightforward guide*. Ed. by Medium. Feb. 25, 2019. URL: https://towardsdatascience.com/fasttext-sentiment-analysis-for-tweets-a-straightforward-guide-9a8c070449a2.

[14] Derrick Mwiti. *Convolutional Neural Networks: An Intro Tutorial*. Ed. by Medium. May 8, 2018. URL: https://heartbeat.fritz.ai/a-beginners-guide-to-convolutional-neural-networks-cnn-cf26c5ee17ed.

[15] Derrick Mwiti. *Convolutional Neural Networks: An Intro Tutorial*. Ed. by Medium. May 8, 2018. URL: https://heartbeat.fritz.ai/a-beginners-guide-to-convolutional-neural-networks-cnn-cf26c5ee17ed.

[16] Rajat Newatia. *How to implement CNN for NLP tasks like Sentence Classification*. Ed. by Medium. May 27, 2019. URL: https://medium.com/saarthi-ai/sentence-classification-using-convolutional-neural-networks-ddad72c7048c.

[17] Vibhor Nigam. *Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning*. Ed. by Medium. Sept. 11, 2018. URL: https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90.

[18] Peter norvig. *How to Write a Spelling Corrector*. Ed. by Norvig. Aug. 1, 2016. URL: https://norvig.com/spell-correct.html.

[19] Martin Pellarolo. *Customers' tweets classification*. Ed. by Medium. Apr. 9, 2018. URL: https://medium.com/@martinpella/customers-tweets-classification-41cdca4e2de.

[20] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Ed. by Medium. Dec. 15, 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[21] Aliaksei Severyn and Alessandro Moschitti. "UNITN: Training Deep Convolutional Neural Network for Twitter Sentiment Classification". In: *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*. Denver, Colorado: Association for Computational Linguistics, June 2015, pp. 464–469. DOI: 10.18653/v1/S15-2079. URL: https://www.aclweb.org/anthology/S15-2079.

[22] Irhum Shafkt. *Intuitively Understanding Convolutions for Deep Learning*. Ed. by Medium. June 1, 2018. URL: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1.

[23] Prajwal Shreyas. *Sentiment analysis for text with Deep Learning*. Ed. by Medium. Apr. 2, 2019. URL: https://towardsdatascience.com/sentiment-analysis-for-text-with-deep-learning-2f0a0c6472b5.

[24] Kristoffer Stensbo-Smidt. *Teaching computers to understand the sentiment of tweets*. Ed. by Medium. Jan. 11, 2019. URL: https://towardsdatascience.com/making-computers-understand-the-sentiment-of-tweets-1271ab270bc7.

[25] Dario Stojanovski et al. "Twitter Sentiment Analysis Using Deep Convolutional Neural Network". In: vol. 9121. June 2015. DOI: 10.1007/978-3-319-19644-2_60.

[26] SuperDataScience. *The Ultimate Guide to Artificial Neural Networks (ANN)*. Ed. by SuperDataScience. Aug. 31, 2018. URL: https://www.superdatascience.com/blogs/the-ultimate-guide-to-artificial-neural-networks-ann.

[27] SuperDataScience. *The Ultimate Guide to Convolutional Neural Networks (CNN)*. Ed. by SuperDataScience. Aug. 28, 2018. URL: https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn.

[28] Duyu Tang, Bing Qin, and Ting Liu. "Deep learning for sentiment analysis: successful approaches and future challenges". In: *WIREs Data Mining and Knowledge Discovery* 5.6 (2015), pp. 292–303. DOI: 10.1002/widm.1171. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1171. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1171.

[29] Thekrutarth. *CNN-for-Twitter-Sentiment-Analysis*. Ed. by GitHub. Feb. 9, 2017. URL: https://github.com/thekrutarth/CNN-for-Twitter-Sentiment-Analysis.