# MA4J5: Sentiment Analysis of Amazon Kindle Reviews with TensorFlow

Ameer Ali Saleem, u2001820

January 6, 2024

**Abstract**

Deep Learning is the subset of Artificial Intelligence pertaining to Machine Learning models which are inspired by the human brain. In this report, we explain the process of training Artificial Neural Networks, using the TensorFlow Python package, to predict the star ratings of Amazon customer reviews. We will elaborate on network layers, model performances and model tuning.

# Contents

# 1 Introduction

## 1.1 Neural Networks and How They Learn

We begin with a brief primer on what neural networks are and how they learn. A perceptron is an individual component of a neural network that will 'fire' if the input exceeds a certain value called the threshold. Concretely, a perceptron $h_{\boldsymbol{w},b}$, parametrised by a weight vector $\boldsymbol{w}$ and a bias value $b$, applies a so-called activation function $g$ to an input $\boldsymbol{x}$. In the simplest case, $g$ is the indicator function $\mathbb{1}$, with

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) = g(\boldsymbol{x}) = \mathbb{1}_{\{\boldsymbol{w}^T\boldsymbol{x}+b>0\}} = \begin{cases} 1 & \boldsymbol{w}^T\boldsymbol{x}+b>0 \\ 0 & \boldsymbol{w}^T\boldsymbol{x}+b\leq 0 \end{cases}$$

where $x \in \mathbb{R}^d$ corresponds to an input of $d$ dimensions. The parameters $\boldsymbol{w}, b$ are learned by the perceptron as part of model training.

The indicator function is just one example of an activation function. Another such example is the sigmoid function, whose formula is given by

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

Generally, we would like our activation functions to be differentiable and monotonic. The differentiability condition enables us to conduct backpropagation on the neural network, which is how the neural network learns.

A neural network is a combination of perceptrons arranged in layers, with connections made between the perceptrons in different layers. A basic example of this is provided in Figure 1. We can define a linear map between layer $k-1$ and layer $k$ as $\boldsymbol{W}^{(k)} : \mathbb{R}^{d_{k-1}} \to \mathbb{R}^{d_k}$, where $d_i$ denotes the number of perceptrons in layer $i$. Taking the activation function across all perceptrons to be the sigmoid function $\sigma$ for simplicity and applying $\sigma$ component-wise, we can write the map from layer $k-1$ to $k$ as $\sigma(\boldsymbol{W}^{(k)}\boldsymbol{x}+\boldsymbol{b}^{(k)})$, where we combine all the biases of layer $k-1$ into a single bias vector. With this, we can define the activation of each layer of an $l$-layer neural network as

$$\begin{aligned} F^{(1)}(\boldsymbol{x}) &= \sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{x}+\boldsymbol{b}^{(1)}\right) \\ F^{(k+1)}(\boldsymbol{x}) &= \sigma\left(\boldsymbol{W}^{(k+1)}F^{(k)}(\boldsymbol{x})+\boldsymbol{b}^{(k+1)}\right), \quad 1\leq k<\ell. \end{aligned}$$

where $\boldsymbol{x}$ is our vector of input values[1]. We can write this recursion more compactly by setting $\boldsymbol{a}^0 \coloneqq \boldsymbol{x}$ and for all $k \in \{2,\ldots,\ell\}$ and dropping brackets in superscripts for convenience, leaving us with

$$\boldsymbol{z}^k \coloneqq \boldsymbol{W}^{(k)}\boldsymbol{a}^{(k-1)}+\boldsymbol{b}^{(k)}, \quad \boldsymbol{a}^k \coloneqq \sigma(\boldsymbol{z}^k).$$

Given an input vector $\boldsymbol{x}$ with corresponding output vector $\boldsymbol{y}$, we aim to tune all the weight and bias parameters in the neural network such that $\boldsymbol{a}^\ell$ is as close to $\boldsymbol{y}$ as possible.

[1]Note that any superscripts wrapped in parentheses are not powers, but rather indicate the value corresponding to the layer. E.g. $\boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$ refer to bias vectors between different layers.

To make this notion precise, we can introduce a loss function $L(\boldsymbol{a}^\ell, \boldsymbol{y}) \in [0, \infty)$, which we leave arbitrary for now. The loss incurred by the neural network depends on the parameter set $(\boldsymbol{W}, \boldsymbol{b})$, where $\boldsymbol{W} = [\boldsymbol{W}^{(2)}, \ldots, \boldsymbol{W}^{(\ell)}]$ is the list of matrix weights and $\boldsymbol{b} = [\boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(\ell)}]$ is the list of bias vectors[2]. This formulation allows us to relate the loss incurred by the network to its parameter set, namely by setting $C(\boldsymbol{W}, \boldsymbol{b}) = L(\boldsymbol{a}^\ell, \boldsymbol{y})$.

One more definition is needed in this section. For each layer $k$ and perceptron $j \in \{1, \ldots, d_k\}$ in layer $k$, we define

$$\delta_j^k \coloneqq \frac{\partial C}{\partial z_j^k}$$

as the sensitivity of the loss function to the input at node $j$ in layer $k$. (From here onwards we use the superscript to specify the layer, with the subscript pointing to the perceptron in that layer.) Denoting $\boldsymbol{\delta}^k \in \mathbb{R}^{d_k}$ as the vector of sensitivities for layer $k$, we have the following result:

**Theorem 1.1 (Backpropagation)** *For a neural network with $\ell$ layers:*

$$\boldsymbol{\delta}^\ell = \sigma'\left(\boldsymbol{z}^\ell\right) \circ \nabla_{\boldsymbol{a}^\ell} L\left(\boldsymbol{a}^\ell, \boldsymbol{y}\right), \tag{1}$$

$$\boldsymbol{\delta}^k = \sigma'\left(\boldsymbol{z}^k\right) \circ \left(\boldsymbol{W}^{(k+1)}\right)^T \boldsymbol{\delta}^{k+1} \tag{2}$$

*where $\circ$ denotes component-wise multiplication. Moreover, the partial derivatives of the sensitivities are given by*

$$\frac{\partial C}{\partial w_{ij}^k} = \delta_i^k a_j^{k-1}, \tag{3}$$

$$\frac{\partial C}{\partial b_i^k} = \delta_i^k. \tag{4}$$

We omit the proof of the above theorem, but it follows analogously from [3, pp. 12-14].

The above work allows us to observe how a neural network is able to learn:

1. **Forward propagation**: We feed the input $\boldsymbol{x}$ into the neural network. By passing the values from layer 1 to layer $\ell$, we obtain the vector $\boldsymbol{a}^\ell$, which is the network's initial output for the input $\boldsymbol{x}$.

2. **Backpropagation**: We now apply Theorem 1.1 to ascertain the sensitivitise of our loss with respect to each weight and bias in the network. Using this, we can apply an algorithm such as gradient descent to alter the weights and biases in such a way that the error incurred by the network is decreased.

3. **Iteration**: we repeat our forward and backward propagation steps until we either attain a sufficiently small error, or we exceed a fixed number of iterations.

[2]Note that the weight matrix indices start from 2 rather than 1, because each matrix represents a linear map from one layer to the next. In comparison, the biases are vectors, so we have one bias vector for every layer.

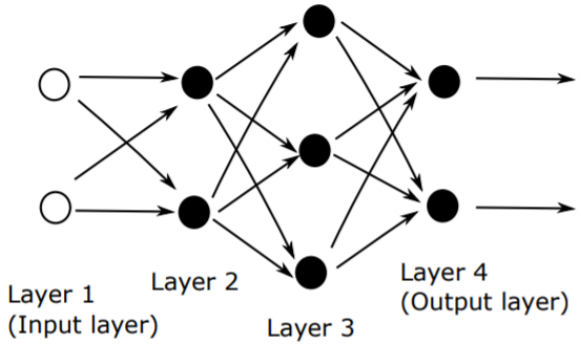



Figure 1: A Neural Network with four layers [3, p. 5].

## 1.2 Natural Language Processing

Natural Language Processing (commonly abbreviated to NLP) is a tract of Artifical Intelligence, devoted to making computers understand the statements or words written in human languages [5, p. 3714].

In order to get a computer to understand the words in a sentence, it is necessary to convert them into vectors in higher dimensional space, with the aim of giving words with similar meaning similar vector representations. For example, we may want to give the words 'lion' and 'leopard' similar vectors in a bank of words containing the names of all the animals in the world. The vector representation of a word is called a word embedding. There are a few choices as to how word embeddings can be achieved, such as with Word2Vec. Word2Vec is not an individual algorithm, but rather a family of model architectures employed for use in NLP-related tasks. Another option is the GloVe framework, which uses a pre-trained vectorised corpus to assign vector values to new words. We discuss in Subsection 1.5 how TensorFlow handles word embeddings.

## 1.3 The Data

Amazon customer review data is publicly available online. The webpage here provides access to a dataset containing customer reviews from 1996 all the way to 2014. My project involves investigating the reviews specifically for Amazon Kindle products, of which there are 5,722,988 records of data. Due to computer processing limitations, I have based my project on the so called '5-core' Kindle review dataset. This subset consists of 2,222,983 reviews, in which each title has 5 reviews present in the dataset. Note that there are plenty of other sectors of Amazon data to choose from.

## 1.4 Data Pre-Processing

We will need to convert our text into computer-recognisable data before feeding it directly into our neural networks. The framework for doing so is as follows:

1. **Text cleaning**: It is important that we remove punctuation and stop words from our text. Stop words are words which do not contribute to the sentiment of a sentence, e.g. 'if', 'me', 'while', etc. If words do not contribute to the sentence's sentiment, then they will only act as noise for our model, which will hinder model

performance. This is best avoided in Machine Learning, especially for Deep Learning tasks such as the one in question.

2. **Tokenisation**: This is the process of converting a sequence of text into individual components called (unsurprisingly) 'tokens'. Tokenisation in its simplest form involves taking each word in a sentence as its own token. More robust tokenisation techniques include N-grams, where every section of N words in a sentence is considered a token. The use of N-grams can help to preserve the context of groups of words in a sentence. For example, the sentence 'This sandwich is not good' indicates a negative sentiment. This can be captured by the 2-gram 'not good', but this compound sentiment would not be recognised if we considered the words 'not' and 'good' in isolation. We have leveraged TensorFlow's built-in tokeniser for this project. This tokeniser builds the word corpus from the data itself, rather than relying on a pre-built corpus.

3. **Padding/Truncating**: Each neural network will expect an input of fixed size. As such, once we have tokenised our input data, we will run into problems if reviews contain different numbers of words. To rectify this, we can add padding (in the form of zeros) to any sentences which do not reach the cap and conversely, can truncate sequences which exceed the limit. Padding/truncating can be done at the start or end of each tokenised review- this is something that one can experiment with.

4. **Train-test-split**: we aim to evaluate model performance on unseen data. As such, we split the dataset into a training set, which the network configures its weights and biases on, and a testing set, where we compare the network's outputs with the true ratings (out of five stars) of each row of text. In this project, we employed a 66%-33% train-test split for all models.

## 1.5 Word Embeddings

Once we have our tokenised reviews, we can project them into higher dimensional space. One way to achieve this is via one-hot encoding. It works as follows [2, p. 14]: suppose that $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ is a discrete random variable. Then the one-hot encoding of any $x_i$ is a vector $\boldsymbol{v}$ where every component of $\boldsymbol{v}$ is zero except for the ith component, which is a 1. For example, assume we have some random variable $\boldsymbol{x}$ that takes values from the set $S = \{a, b, c\}$. Set $x_1 = a, x_2 = b, x_3 = c$. Then the one-hot encoding for $\boldsymbol{x}$ is $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$. Although simple to implement, with an efficient runtime complexity, the main drawback of one-hot encoding in this context is that the one-hot vectors do not provide any information about how similar/different two output vectors are. Also, the embedding vectors are sparse: with $n - 1$ zeroes in each vector, one cannot help but wonder if there is a more efficient way of embeddings the word tokens.

One way to quantify this vector similarity is by the cosine similarity: given word embeddings $\boldsymbol{A}$ and $\boldsymbol{B}$, the cosine similarity between the corresponding words is

$$\cos(\theta) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{||\boldsymbol{A}|| \cdot ||\boldsymbol{B}||}$$

This enables us to compare the angle between vectors, with a close-to-zero value indicating high similarity. In any case, dense vectors make better use of the dimensions of

each vector compared to one-hot encoding.

TensorFlow's embedding layers provide a more robust approach than either of the above. These layers will produce dense vectors of floating point values, whose composition is derived from weights which the neural network can learn, e.g. through the steps outlined in Subsection 1.1. This allows for greater flexibility in vector assignment which is contingent on the training data itself, rather than from an arbitrary embedding vector space.

# 2 Overview of Neural Network layers in TensorFlow

## 2.1 Neural Network Layers

In the below section, we outline some of the more popular neural network layers for NLP and beyond:

- **Embedding layer**: These TensorFlow layers have been explained in Subsection 1.5.
- **Pooling layer**: Pooling methods aim to reduce the size of the input space by replacing groups of points with single points formed by aggregating the groups' values. As illustrated in Figure 2, the two most common types of pooling methods are: max pooling, in which the maximum value of a group of points is taken as the value of the new aggregated point; and average pooling, where the average value is taken instead. In general, max pooling is able to highlight stark contrasts in a dataset, whereas average pooling is better for 'smooothening out' the values in the entirety of the dataset. In this project, pooling layers can be applied immediately after an embedding layer, reducing the large dimension of word embeddings down to a more manageable size.
- **Convolution layer**: A convolution is a filter applied to an input, with multiple filters constituting a feature map. Convolution layers are often used in tandem with pooling layers: convolutions create multiple feature channels in parallel; and the pooling layer reduces the dimension of each of these channels.
- **Dropout layer**: This layer randomly sets activation values to 0, temporarily 'dropping out' the corresponding nodes from the network. The main purpose of dropout layers is to help prevent model overfitting. This layer is especially useful when training a large neural network on a relatively small dataset.
- **Long Short-Term Memory (LSTM)**: LSTM networks are a specific example of the Recurrent Neural Network architecture. The idea behind an RNN is to remember the previous state of the network. However, in the context of NLP, it may be important to remember words that came before the previous state. The solution for such a problem was the LSTM network. The mathematics behind how LSTM works is beyond the scope of this report, but one can read [4] if interested in learning more.

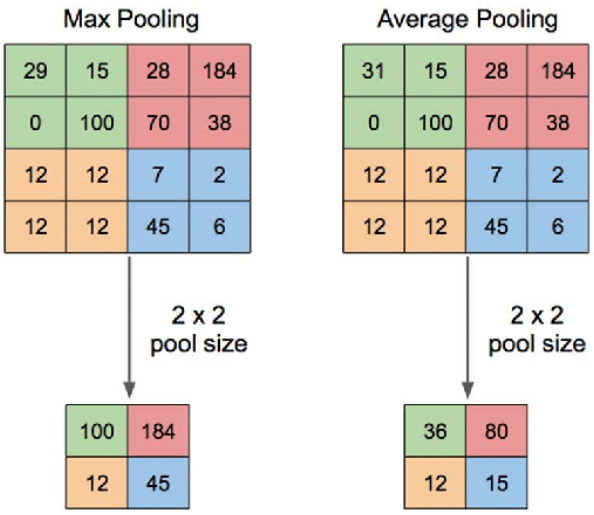



Figure 2: Illustrations of max pooling and average pooling. [Source: ResearchGate.]

## 2.2 Activation Functions and Loss Metrics

In Subsection 1.1, we introduced the concept of an activation function. Activation functions are applied to the expression $\boldsymbol{w}^T\boldsymbol{x}+b$ and are commonly used for scaling inputs and providing an avenue of differentiability in which gradient descent to be leveraged. Our NLP problem is a classification problem: given a text review, predict whether the review was attributed a score of one, two, three, four or five stars. To indicate a four star review label, we could transform the output 4 into the one-hot encoded vector [3] $\boldsymbol{y} = (0, 0, 0, 1, 0)$. We could then allow our network output to be a vector of probabilities. So the output $\hat{\boldsymbol{y}} = (0.1, 0.6, 0, 05, 0.25)$ indicates a moderate prediction of a two-star review. In order to obtain $\hat{\boldsymbol{y}}$, we utilise the softmax activation function

$$p_i(\boldsymbol{x}) := \frac{e^{x_i}}{\sum_{j=1}^{5} e^{x_j}},$$

where $\boldsymbol{x}$ is the final neural network output. It is clear that $\sum_{i=1}^{5} p_i = 1$, as expected [4]. In particular, setting $\hat{\boldsymbol{y}} = \boldsymbol{p}(\boldsymbol{x})$ provides the desired network output.

It was also in Subsection 1.1 that we discussed the idea of a loss function. In order to compare $\hat{\boldsymbol{y}}$ against the true $\boldsymbol{y} = (0, 0, 0, 1, 0)$, we appeal to the categorical cross-entropy loss function, given by

$$CCE = -\sum_{i=1}^{5} y_i \cdot \log \hat{y}_i.$$

This framework will reward the network for high confidence in the correct class and low confidence in incorrect classes, and penalise it in the converse cases. The softmax activation function and the categorical cross-entropy loss function are standard for multiclass

[3] Note that the one-hot encoding we are discussing here is not for word embeddings as discussed in subsecton 1.5, but rather for the network output values.

[4] Note that when $n = 2$, the softmax function reduces down to the Sigmoid function.

classification problems. We are now left with the task of determining what TensorFlow networks to experiment with.

# 3 Five-Category Classification

Given a dataset of Amazon Kindle customer reviews, can we a train a neural network to predict the star rating out of 5?

Before following any of the data pre-processing steps outlined in section 1.4, we first observe the spread of review scores in the dataset. Training a neural network with this imbalance will lead to biased training: when unsure of how to evaluate an incident review, the network will default to assigning a large softmax value to the most prevalent star rating, because the network will have been exposed to far more ratings of this class than any other. Figure 3 demonstrates the sheer imbalance of classes in the dataset, as well as the result of applying undersampling to the data.

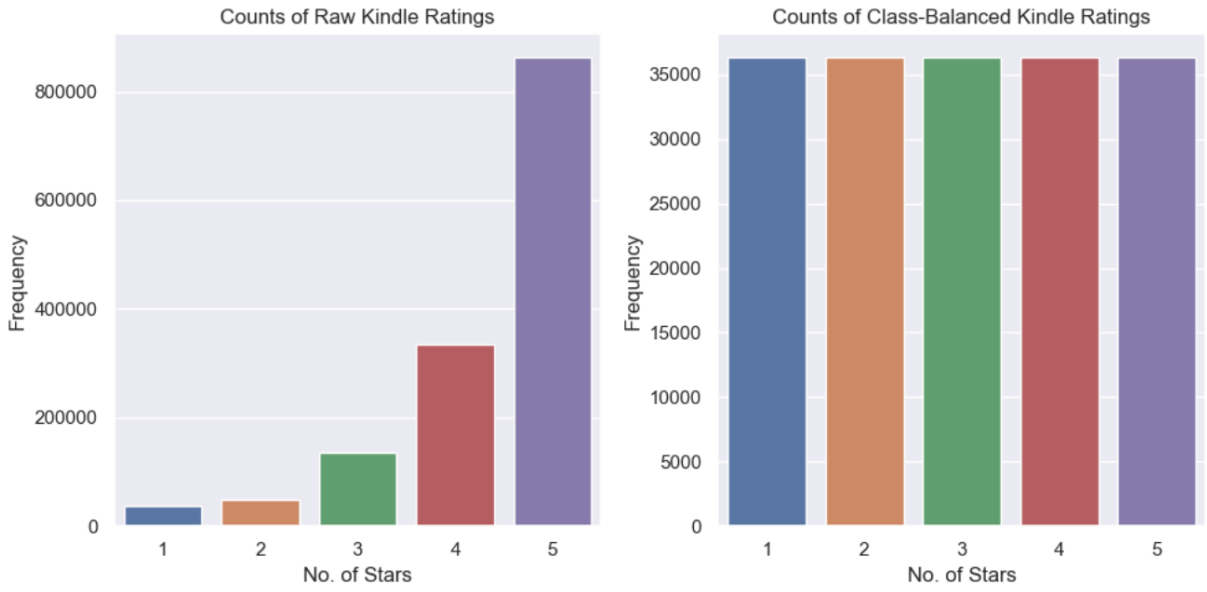



Figure 3: Dealing with class imbalance.

## 3.1 Parameters and Hyperparameters

The tokenisation process allows for the selection of the following parameters. The variable names correspond to the names provided in the relevant Python notebook:

- **stopwords**: The Natural Language ToolKit module in Python contains its own list of standard stopwords. From experimentation with model training, it was found that this list was too restrictive; the list also contains words like 'not' and 'isn't' which are very important when evaluating the sentiment of phrases like 'This book was not good' or 'This read isn't a good use of your time'. A such, we used only a subset of NLTK's stopwords for text cleaning.
- **num_words**: TensorFlow's tokeniser builds the word corpus using the data in the training set. A corpus of size 20,000 seemed to be a good fit. This means that the

top 20,000 most common words in the entire training set were attributed a value. The remaining words were all then assigned the default value ' <OOV> ' (Out Of Vocab).

- **max_length**: When each review is tokenised, the words are replaced with numbers. We must ensure that each tokenised sequence is of the same length, as each neural network will expect inputs of a fixed size. Hence, we specify a cap on the length of each sequence. Through experimentation, a cap of 50 seemed to work well with the data.

- **padding/truncating**: As mentioned in Subsection 1.4, one can experiment with the position of padding and truncating. Through experimentation with our Amazon review models, it was found that modifying these parameters had no noticeable impact on model performance.

The neural network architectures also had their own hyperparameters. Hyperparameters are parameters that are not learned during the network training process, but rather are set by the user. We outline below the hyperparameters common across all the architectures used:

- **output_dim**: Each of our networks starts by taking a tokenised review and projecting it into higher dimensional space. The dimensionality of this output space is determined by the value of 'output_dim'. Most of our networks leverage embedding dimension 16.

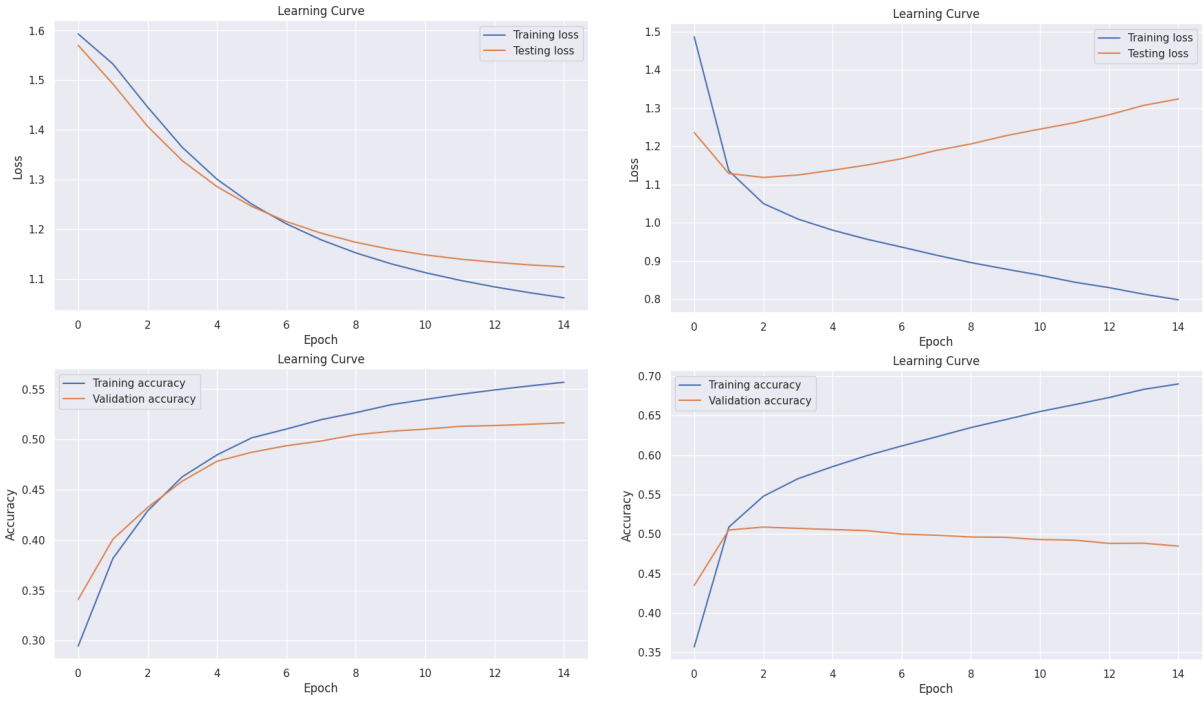
- **epochs**: Networks tend to benefit the most when trained on multiple iterations of the entire training set. Each training iteration over the training set is referred to as an 'epoch'. Training with an insufficient number of epochs will hinder the network's learning capabilities, resulting in model underfitting. Conversely, training over too many epochs may cause the model to overfit the training data. Due to computational limitations, we have fixed all our networks to learn across 15 epochs.

- **batch_size**: If the process of forward propagation and backpropagation (outlined at the end of Subsection 1.1) was conducted on the entire training set for each epoch, then the model would most likely struggle to capture the finer details in the data. To remedy this, we apply the forward/backward propagation process using smaller batches of the training set each time. This helps the model learn more from individual reviews, but causes training time to increase. As such, optimising the batch size comes down to the trade-off between accuracy and speed. In any case, each epoch is comprised of enough batches to constitute the entire training set. We have opted with a batch size of 1000 samples for our network training.

- **optimizer**: TensorFlow allows for the user to choose the algorithm to be applied when optimising the stochastic gradient descent method in the backpropagation step. The most popular choice of optimiser in Deep Learning problems appears to be the Adam algorithm, which was proposed in 2015 and can be read more about in [6]. We will use this optimiser for all our models.

## 3.2 Model Performance and Activation Functions

Model performance is governed by two metrics: loss and accuracy. Across epochs, we hope to see network loss decrease and accuracy increase. As described in 2.2, we will use categorical cross-entropy as our loss function. Accuracy is given by the proportion of correctly classified instances. Optimal learning is best indicated when the loss and accuracy values are similar across both the training and testing sets. Low training loss and high testing loss indicates model overfitting, which we aim to avoid. Examples are provided in Figure 4.




(a) The loss curves are close to each other and relatively smooth, indicating a good fit.

(b) The training loss eventually begins increasing, indicating model overfitting.

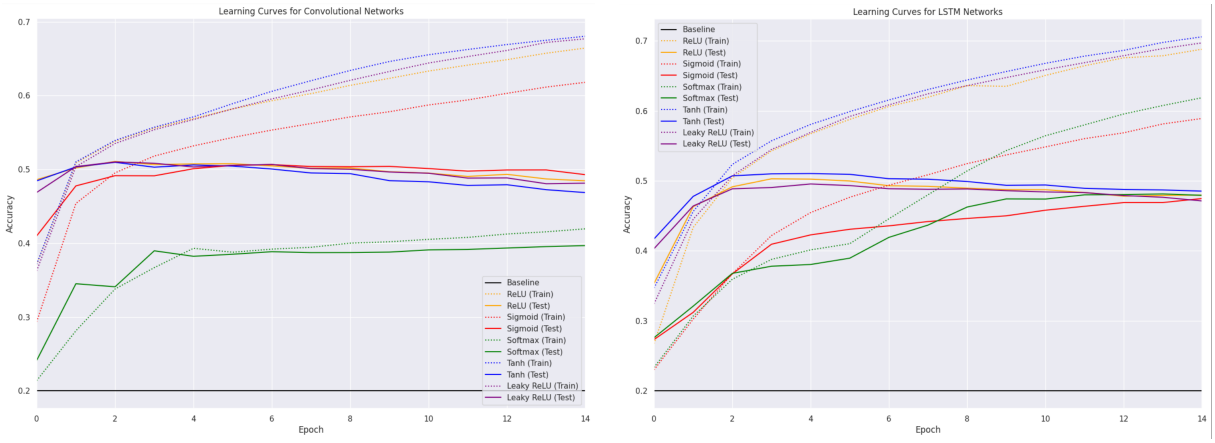
Figure 4: Example learning curves: row 1 provides loss curves and row 2 provides accuracy curves.

Before exploring different neural network architectures, we first compare different network layers to ascertain the optimal choice of activation function, if such an optimal choice exists for our data. This is achieved by fixing standard choices of neural network architecture, modifying the activation function involved and comparing the results. We fixed a convolutional network architecture and an LSTM network architecture. The learning curves across different activation functions are provided in Figure 5.

We observe that the models achieved roughly similar accuracies across the different activation functions selected. As such, it appears that our choice of activation function does not make much difference in model performance.

## 3.3 Results

Figure 6 illustrates the accuracies achieved by various network architectures. A classifier which makes the same prediction for every review can only achieve 20% accuracy in




(a) Convolutional networks

(b) LSTM networks

Figure 5: Accuracy curves for neural networks across different activation functions.

our class-balanced data. Despite experimenting with many different layers, the models struggled to exceed accuracy values above 50%. The code for the two best network architectures are displayed in Figure 7. The capability for even the simpler networks to overfit the data was surprising. The low performance of the models may be due to the fact that, as we are dealing with a five-category classification, we have ambiguity between adjacent classes. That is, it can be difficult for a person to distinguish between a five-star review and a four-star review, let alone a computer.

To investigate the results further, confusion matrices were constructed to visualise the trends in model prediction. The matrices for our two best models (the pooling layer model and the LSTM model, which demonstrated the least signs of overfitting) are shown in Figure 8. The patterns across both matrices demonstrate the effectiveness of training; ideally we would like larger values on the main diagonal of each matrix. We certainly have many cases in which the models made, say, a 5-star prediction when, in reality, the actual rating was only 4-stars. Interestingly, both models appeared more confident in the most extreme cases, with 1-star and 5-star predictions being the most commonly predicted categories.

A more granular way to investigate the performance of these two models is to compare their predictive confidence distributions. These are depicted in Figure 9. The x-axis of these plots contains the actual review ratings, and the bar charts display the average softmax outputs of the models. We observe that the LSTM model demonstrates greater confidence in its classifications than the pooling model; the red bar in the '1 Star' category has a larger value in 9b than 9a, the green bar in the '2 Star' category has a larger value 9b than 9a, etc. This makes sense, given that the LSTM framework is more robust, especially for NLP problems. However, from the learning curves in Figure 6, we know that the LSTM model overfits the data more than the pooling model, so this confidence may not be entirely sound.

Figure 9 also provides some explanation as to why our models are more confident in predicting the extreme categories: the '1 Star' category has only one immediate neighbour (the '2 Star' category), and similar can be said for the '5 Star' category. However, the '3 Star' category has two immediate neighbours ('2 Star' an '4 Star') and so misclassification can be conducted more easily either side of this category.

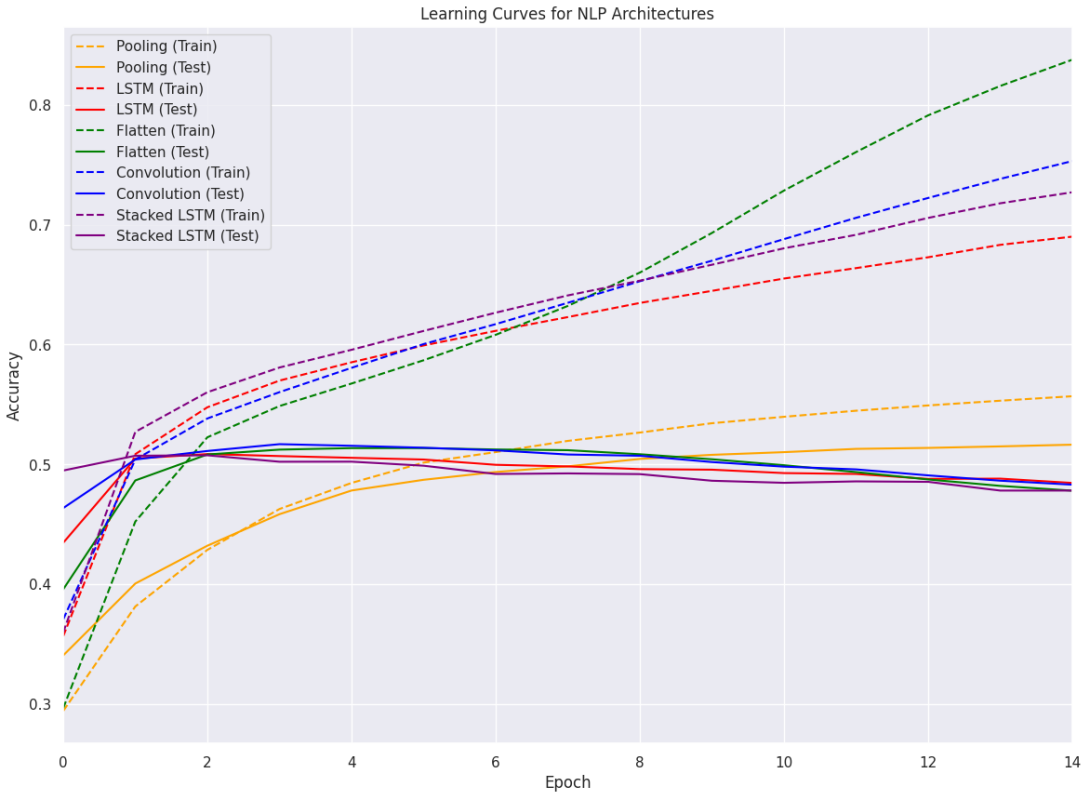



Figure 6: Accuracies achieved by different network architectures.

# 4 Simpler Classification Tasks

Achieving high accuracies in a five-category classification problem is a challenging task in Natural Language Processing. As such, we perturb our original problem slightly to see whether greater accuracy can be achieved in a simpler problem setup. In particular, how do our networks perform when predicting reviews as either 'positive' (combination of 4 & 5 star reviews), 'neutral' (3 star reviews) or 'negative' (combination of 1 & 2 star reviews)?

After dealing with class imbalance once again, the same models from the previous section were trained for this simpler classification task. The learning curves are depicted in Figure 10. The validation accuracy of each model is roughly 70%. The models' overfitting potential is significantly less than in the original five-category problem.

Finally, we explore what happens when only 'positive' and 'negative' reviews are considered, i.e. removing the 'neutral' category from the previous subsection. In this case, our models are able to achieve 90% accuracy, as shown in Figure 11. Once again, the pooling model avoids overfitting the best, although all the models perform very well.

# 5 Limitations and Improvements

We observed in the original 5-category problem that models performance peaked at roughly 50%, despite many different model architectures and parameters being experimented with. However, this increased to 70% in the three-category problem and even to 90% in the binary setup. These increases in accuracy exemplify the correctness of

```python
# model hyperparameters
embedding_dim = 16
num_epochs = 15
batch_size = 1000

# Average pooling
model1 = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(5, activation='softmax')
])
model1.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```

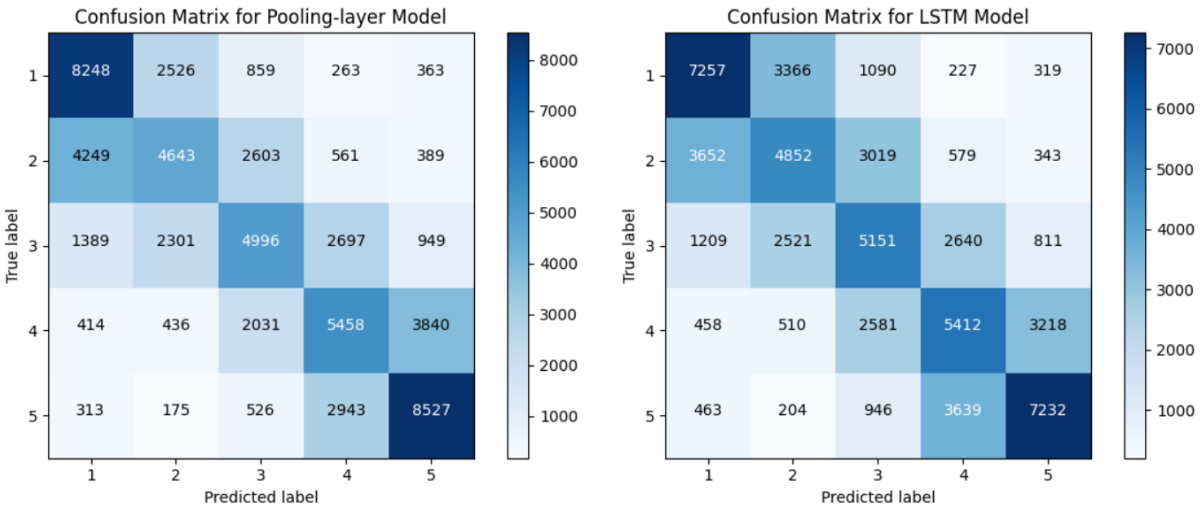
(a) Pooling network code

```python
# model hyperparameters
embedding_dim = 16
num_epochs = 15
batch_size = 1000

# LSTM setup
model2 = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(8)),
    tf.keras.layers.Dense(8, activation='tanh'),
    tf.keras.layers.Dense(5, activation='softmax')
])
model2.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```

(b) LSTM network code

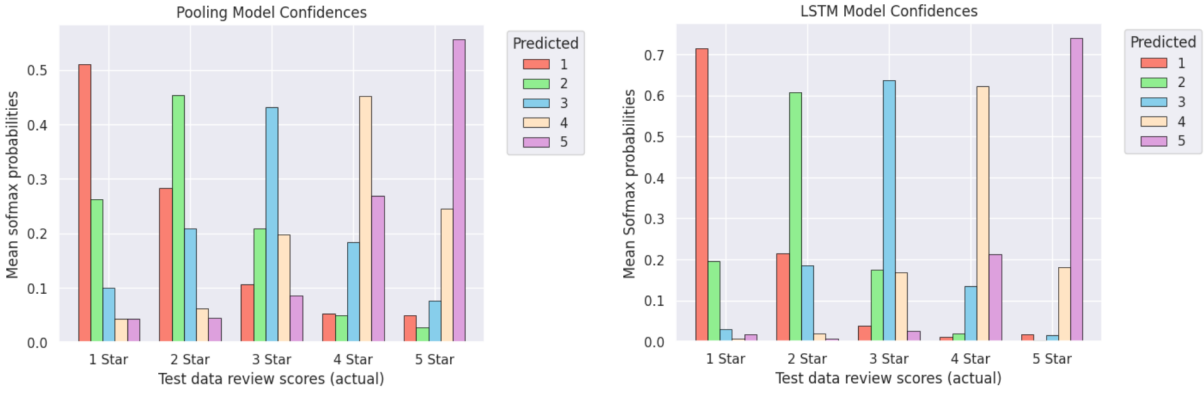
Figure 7: Python code for the two best performing networks in the 5-category classification problem.




(a) Pooling model classifications

(b) LSTM model classifications

Figure 8: Confusion matrices for best performing models.




(a) Pooling model confidences

(b) LSTM model confidences

Figure 9: Predictive confidence distributions of top 2 models.

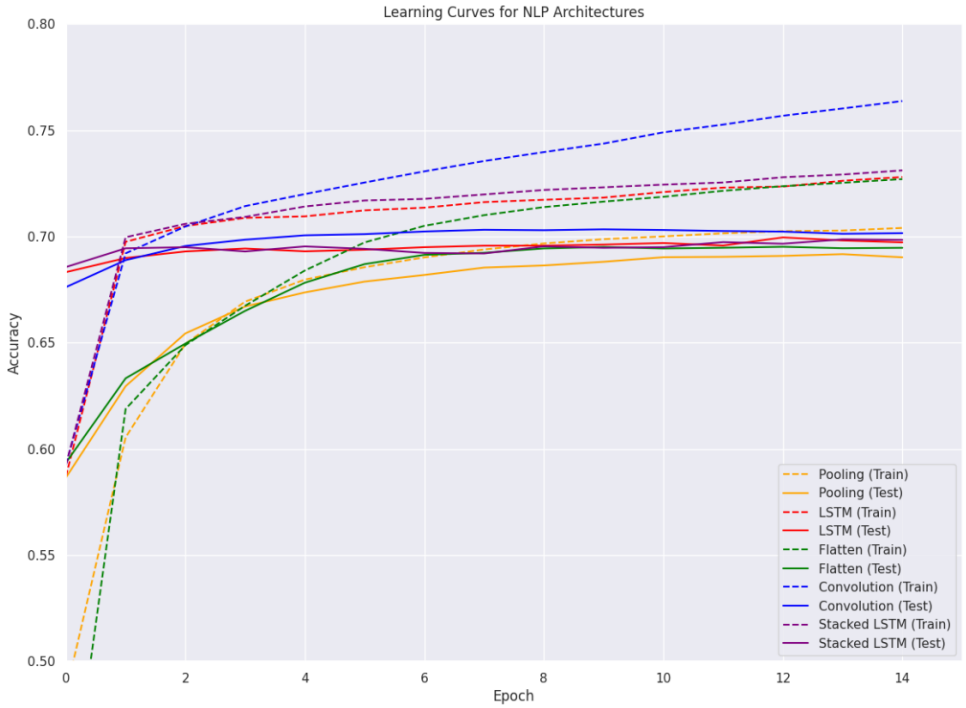



Figure 10: Accuracies achieved in the three-class problem.

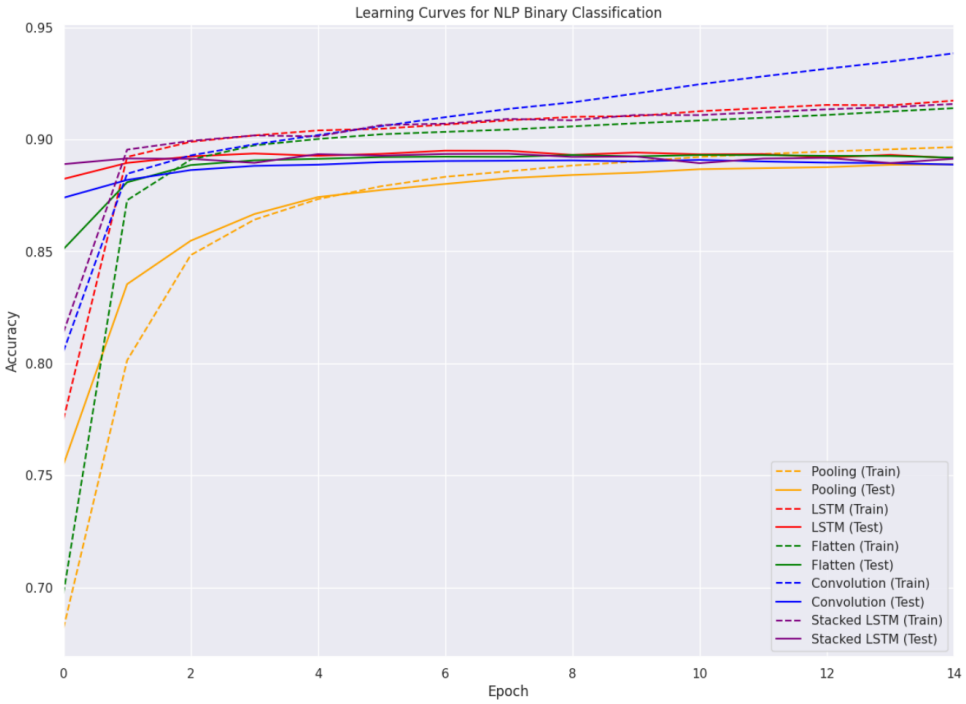



Figure 11: Accuracies achieved in the binary-class problem.

our models, but highlight limitations elsewhere. The dataset itself could pose one such limiting factor. As mentioned earlier in this report, the distinction between, say, a 5-star review and a 4-star review is difficult to discern, as different people may have different standards as to what a 5-star review constitutes. Classification can become especially more difficult in the case of ambiguous review texts; NLP models generally struggle with sarcastic text, for example. A technical decomposition of how sarcasm detection can be handled can be found in [1].

It can be observed in Figure 6 that most of our models' validation accuracies start to plateau and even decline. One way to rectify this is by setting the number of epochs dynamically: we can instruct the models to terminate the learning process once the accuracy on the test set begins to decrease. The choice was made not to do this for this project- the aim was to compare models when they have all been trained for the same number of epochs. Finally, there is the issue surrounding model tuning. There are many parameters and hyperparameters associated with our models as described in Subsection 3.1. While experimentation was conducted with some of these, it would prove rather difficult to optimise model performance over *all* the available (hyper) parameters in a student project such as this. And out of all the (hyper) parameters experimented with in this project, the more optimal choices did not correspond to sizeable boosts in model performance.

# References

[1] Ashwitha A, Shruthi G, Shruthi H R, Makarand Upadhyaya, Abhra Pratip Ray, and Manjunath T C. Sarcasm detection in natural language processing. *Materials Today: Proceedings*, 37:3324–3331, 2021. ISSN 2214-7853. doi: https://doi.org/10.1016/j.matpr.2020.09.124. URL `https://www.sciencedirect.com/science/article/pii/`

S2214785320368164. International Conference on Newer Trends and Innovation in Mechanical Engineering: Materials Science.

[2] John Hancock and Taghi Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7, 04 2020. doi: 10.1186/s40537-020-00305-w.

[3] Catherine F Higham and Desmond J Higham. Deep Learning: An Introduction for Applied Mathematicians. *Siam review*, 61(4), 2018.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[5] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: state of the art, current trends and challenges. *Multimedia Tools and Applications*, 82(3):3713–3744, July 2022. ISSN 1573-7721. doi: 10.1007/s11042-022-13428-4. URL http://dx.doi.org/10.1007/s11042-022-13428-4.

[6] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2015.