Alden_Andrew_01666104

anonymous marking enabled

Submission date: 06-Sep-2021 08:24PM (UTC+0100)
Submission ID: 159528944
File name: Alden_Andrew_01666104.pdf (1.17M)
Word count: 19097
Character count: 91320

Imperial College London

Imperial College London

Department of Mathematics

Natural Language Processing for Financial Chat Message Classification

Author: Andrew Alden (CID: 01666104)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2020-2021

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Acknowledgements

I would like to express my gratitude towards my supervisor Dr. Antoine Jacquier for his invaluable advice and constant support.

I would also like to thank the members of the Global Markets Lab at BNP Paribas. In particular, I would like to thank Juyuan Liu whose guidance and feedback were indispensable in the preparation of this thesis.

I would also like to thank my parents and my sister for their encouragement and support throughout my studies.

Abstract

This thesis focuses on applying Natural Language Processing (NLP) techniques to classify chat messages exchanged amongst professionals working in financial institutions. As messages are constantly being exchanged amongst organisations and their stakeholders, analysing these messages will give organisations a better understanding of current topics that are generating interest.

The message classifier used for the purposes of this research uses the Bidirectional Encoder Representations from Transformers (BERT) model to encode the messages. A classification head consisting of a bi-directional Long Short-Term Memory (LSTM) model and multiple Feedforward Neural Networks (FNN) is built on top of BERT to classify the data. The various aspects which form part of the model are described in detail. Techniques to improve model performance and speed up training are presented. The model is tested on a dataset comprising of instances obtained using two text sources. Results showed that the classifier performed well. Moreover, the classifier is compared to a simpler model which uses Support Vector Machines (SVM). Finally, context is added to the messages in the dataset to try and improve the results.

Contents

List of Figures

List of Tables

Introduction

The amount of unstructured data is rapidly increasing as a result of technological advancement. Approximately 90% of new data generated daily is unstructured and "*unstructured data is growing at* 55-65 *percent each year*" (Marr [26]). Text data is one of the most abundant forms of unstructured data. To extract meaningful information from text data as well as to keep up with the pace at which new data is being generated, computer algorithms are needed. For this reason, multiple Natural Language Processing (NLP) techniques have been developed and applied to different tasks.

As messages are constantly being exchanged amongst organisations and their stakeholders, analysing these messages will give organisations a better understanding of current topics that are generating interest. For example, in the case of financial institutions, these being the focus of this research, analysing these messages will assist in identifying sales trends and in influencing trading decisions.

The model used to classify these messages is split into two parts; an encoder and a classification head. To feed the messages into a classifier, the messages need to be converted into vectors. These vectors are known as word embeddings, and the algorithm used to construct these word embeddings is a crucial aspect of the model. The Bidirectional Encoder Representations from Transformers (BERT) is used to construct the embeddings. BERT is a pre-trained model which is fine-tuned to specific downstream tasks. In our case the downstream task is multi-label classification. The embeddings are the input to a classification head. The classification head consists of a bi-directional Long Short-Term Memory (bi-LSTM) model. Using a bi-LSTM model allows us to consider words in the forward direction as well as in the

reverse direction, providing a better understanding of the content of the message. The output of the bi-LSTM is fed as input to Feedforward Neural Networks (FNN), each FNN acting as a binary classifier for a particular label.

In chapter 1 we describe the main methods used for text classification as well as the evolution of word embeddings. In chapter 2 we concentrate on the models used. Namely, we focus on FNNs, Recurrent Neural Networks (RNN), Transformers, and BERT. Chapter 3 addresses model training, regularization, the dataset, and the results obtained.

Chapter 1

Introduction to Text Classification and Word Embeddings

In order to perform text classification, the text data will need to be embedded into a vector space. The method used to map a text sequence to vectors is a fundamental aspect of most NLP models. A basic model used to construct embeddings is the bag of words (BOW) model (Eisenstein [10, Chapter 2 *Linear Text Classification*, pages 13-16]). In the BOW model, we first start by specifying a vocabulary (a list of words). Each text instance is represented as the vector $\boldsymbol{x} = (x_1, \cdots, x_V) \in \mathbb{R}^V$ where x_i is the number of times the i^{th} word in the vocabulary appears in the current text sequence and V is the vocabulary size. Using this representation, the vector $\boldsymbol{x}$ contains information on the count of each word. However, it does not contain any other useful information, such as grammar, sentence boundaries, etc. Another limitation of this encoding is its high dimensionality. Whenever a BOW model is used, the embedding's dimension is equal to the size of the vocabulary which can be very large. Suppose we want to perform multiclass classification consisting of the labels $\mathcal{Y}$. For each label $y \in \mathcal{Y}$, we compute a score $\Phi(\boldsymbol{x}, y)$. This score is a compatibility function between the text representation and the label y. It is computed using a weights matrix $\boldsymbol{\theta}$, where $\boldsymbol{\theta}_{i,j}$ scores the compatibility between the i^{th} word in the vocabulary and the j^{th} label. The label $\hat{y}$ is chosen such that

$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \, \Phi(\boldsymbol{x}, y) \, .$$

An extension of the BOW model is the n-gram model. An n-gram model takes into account multi-words as features. For example "ref curve" and "bank of england" are examples of a bigram (2-gram) and a trigram (3-gram) respectively. In the n-gram model, each sequence of n words is taken as *one* word in the BOW model. Hence, if $n = 1$ we get the classical BOW model. As n increases, the text representation dimension increases and generalisation decreases as new text passages will not have many n-grams in common with the training data. This is an example of the classical bias-variance trade-off encountered in machine learning. Moreover, the n-gram model suffers from the same drawbacks as the BOW model; high-dimensionality and minimal information encoded into the representation (only frequency is encoded).

To address some of the drawbacks associated with the above models (and variants thereof), representational learning is adopted to construct word embeddings. Representational learning involves building models which *learn* a representation of the instances in the text dataset. One of the most popular models is the *Word2Vec* model (Mikolov et al. [27]). There are two models associated with Word2Vec; Continuous Bag of Words (CBOW) and Skip-gram. Both models learn an embedding for words in the vocabulary in an unsupervised way. They differ in the way they tackle the unsupervised learning. The CBOW model uses neural networks to predict a centre word given the $2h$ surrounding words. If the centre word is x_m, the inputs to the model are $x_{m-h}, \cdots, x_{m-1}, x_{m+1}, \cdots, x_{m+h}$. To learn the centre word x_m, context is being taken into account. On the other hand, in the skip-gram model the objective is to learn the words $x_{m-h}, \cdots, x_{m-1}, x_{m+1}, \cdots, x_{m+h}$ given the centre word x_m. This trains the model to predict the context from the centre word. The Word2Vec model captures local information in its embeddings as it learns from the surrounding words. An embedding which captures global and local information is the Global Vector (GloVe) embeddings (Pennington et al. [30]). Word co-occurrences in the training data are used to encapsulate local and global statistics in the embeddings.

After using a model to construct word embeddings, these embeddings are then fed into another model used to perform classification. An inherent feature of text data is

its sequential nature. Therefore, models which are capable of extracting meaningful relationships between sequential data as input and labels are ideal for text classification. RNNs provide this functionality and hence are prominent in the field of NLP. In a RNN, the output from the previous state is fed back into the system as input to the current state, along with additional input. In terms of text data, RNNs are advantageous as they can condition on previous words in the sequence. Moreover, bi-directional RNNs condition on past as well as future words. This is generally preferred whenever the whole text sequence is known in advance. A drawback of RNNs is that they encounter difficulties in capturing long-term dependencies. This could affect model performance, especially when needing to condition on previous words in a text sequence spanning multiple sentences. To alleviate this, the Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) are used.

Recently, most advancements in NLP have been in the area of pre-trained language models. A pre-trained model is a model trained using general tasks and is then fine-tuned to a downstream task. ELMo (Peters et al. [31]) is a deep contextualized word representation model which captures syntax, semantics, as well as polysemy in its word embeddings. This is achieved by pre-training bi-LSTMs. Another breakthrough in NLP is the Transformer (Vaswani et al. [38]). The Transformer is a model which does not use recurrence and relies on attention mechanisms. The Transformer consists of two main sections; an encoder and a decoder. The encoder constructs the word embeddings and the decoder retrieves words from their embeddings. A modified version of the Transformer's decoder is used in the OpenAI Generative Pre-Trained Transformer (GPT) (Radford and Narasimhan [33]). Another model based on the Transformer is BERT (Devlin et al. [8]). BERT uses the encoder section of the Transformer and it is a pre-trained deep bi-directional representational model which jointly conditions on both the left and right context in all layers to construct word embeddings.

Chapter 2

Message Classifier

2.1 Feedforward Neural Networks

The objective of most machine learning tasks is to construct a function $\boldsymbol{f}: \mathbb{R}^I \to \mathbb{R}^N$ which maps inputs to outputs in an optimal way. Classical machine learning algorithms are suitable for capturing linear and certain non-linear relationships between inputs and outputs, however they struggle in capturing very complex relationships. FNNs are a class of non-linear functions which can represent most types of functional relationships between a set of inputs and a set of outputs. FNNs are an integral part of the word embeddings model incorporated in the message classifier used for the purposes of this research. In this section we describe FNNs and how to train them.

Definition 1. Let $I, N, r \in \mathbb{N}$. Also, let $d_i \in \mathbb{N}$ be the number of units in the i^{th}-hidden layer for any $i = 1, \cdots, r-1$, $d_0 = I$, $d_r = N$, and $\boldsymbol{\sigma}_i: \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$ for any $i = 1, \cdots, r$ be activation functions.

A function $\boldsymbol{f} \in \mathbb{R}^I \to \mathbb{R}^N$ is a FNN with $r-1$ hidden layers if

$$\boldsymbol{f} = \boldsymbol{\sigma}_r \circ \boldsymbol{L}_r \circ \cdots \circ \boldsymbol{\sigma}_1 \circ \boldsymbol{L}_1 \ ,$$

where $\boldsymbol{L}_i: \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ for any $i = 1, \cdots, r$ is an affine function parameterised by weight matrix $\boldsymbol{W}^i = \left[W^i_{j,k}\right]_{j=1,\cdots,d_i,\ k=1,\cdots,d_{i-1}} \in \mathbb{R}^{d_i \times d_{i-1}}$ and bias vector $\boldsymbol{b}^i =$

$\left(b_1^i, \cdots, b_{d_i}^i\right)$, i.e.

$$\boldsymbol{L}_i = \boldsymbol{W}^i \boldsymbol{x} + \boldsymbol{b}^i, \quad \boldsymbol{x} \in \mathbb{R}^{d_{i-1}}.$$

The class of functions $\boldsymbol{f}$ is denoted by

$$\mathcal{N}_r \left(I, d_1, \cdots, d_{r-1}, N; \boldsymbol{\sigma}_1, \cdots, \boldsymbol{\sigma}_r\right).$$

The natural numbers $r, d_1, \cdots, d_{r-1}$ are the hyper-parameters of the FNN, the weight matrices $\boldsymbol{W}^1, \cdots, \boldsymbol{W}^r$ and the bias vectors $\boldsymbol{b}^1, \cdots, \boldsymbol{b}^r$ are the parameters of the FNN, and together with the activation functions they form the architecture of the FNN.

Graphically, the units of a FNN correspond to vertices in a graph and the weights correspond to edges in the graph. Therefore, $W_{j,k}^i$ can be interpreted as the strength at which the output from the k^{th} unit of the $(i-1)^{\text{th}}$ layer is transmitted to the j^{th} unit of the i^{th} layer. Figure 2.1 depicts a FNN belonging to $\mathcal{N}_3\left(4, 6, 6, 2\right)$.

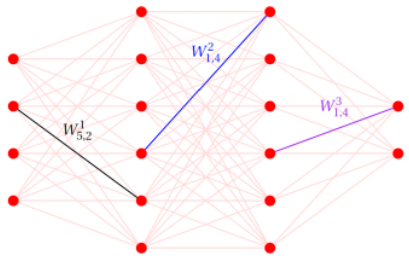


Figure 2.1: Graphical Representation of a FNN in the class $\mathcal{N}_3\left(4, 6, 6, 2\right)$. The edges corresponding to the weights $W_{5,2}^1, W_{1,4}^2$, and $W_{1,4}^3$ are highlighted (Pakkanen [29, Section 1 *Introduction*, Figure 1, page 8]).

One of the main reasons FNNs have become increasingly popular to tackle various machine learning problems is their ability to approximate any function conforming to certain regularity conditions arbitrarily well. This property is called the *Universal Approximation Property* of FNNs and is articulated in Theorem 4.

Definition 2. Let $\boldsymbol{f}$ be a function defined almost everywhere with respect to a Lebesgue measure μ on a measurable subset $\Omega \subseteq \mathbb{R}^n$. $\boldsymbol{f}$ is said to be essentially bounded on Ω if $|\boldsymbol{f}\left(\boldsymbol{x}\right)|$ is bounded almost everywhere on Ω. This is denoted by $\boldsymbol{f} \in L^\infty\left(\Omega\right)$.

Definition 3. Let $\boldsymbol{f}$ be a function defined almost everywhere with respect to a Lebesgue measure μ on a measurable subset $\Omega \subseteq \mathbb{R}^n$. $\boldsymbol{f}$ is said to be locally essentially bounded on Ω if for every compact set $K \subseteq \Omega$, $\boldsymbol{f} \in L^\infty(K)$. This is denoted by $\boldsymbol{f} \in L^\infty_{loc}(\Omega)$.

Let

$$M \coloneqq L^\infty_{loc}(\mathbb{R}) \cap \{f : \mathbb{R} \to \mathbb{R} \mid \mu\left(\overline{A(f)}\right) = 0\}$$

where $\overline{A(f)}$ denotes the closure of the set of discontinuities of f and μ is the Lebesgue measure.

Theorem 4. *Universal Approximation Property:*

(i) Let $\sigma \in M$. Define

$$\mathcal{S}_n \coloneqq span\left\{\sigma(\boldsymbol{W}\boldsymbol{x} + b) : \boldsymbol{W} \in \mathbb{R}^n, b \in \mathbb{R}\right\}.$$

Then $\mathcal{S}_n$ is dense in $C(\mathbb{R}^n)$ iff σ is not an algebraic polynomial almost everywhere (Leshno et al. [24, Section 4 Results, Theorem 1, page 863]).

(ii) Let θ be a non-negative finite measure on $\mathbb{R}^n$ with compact support and absolutely continuous with respect to the Lebesgue measure. Then $\mathcal{S}_n$ is dense in $L^p(\theta)$, $1 \leq p < \infty$, iff σ is not a polynomial almost everywhere (Leshno et al. [24, Section 4 Results, Proposition 1, page 863]).

A proof of Theorem 4 is provided in Leshno et al. [24, Section 6 *Proofs*, pages 864-866].

2.1.1 Training FNNs

Loss Function

FNNs $\boldsymbol{f} \to \mathbb{R}^I \to \mathbb{R}^N$ map inputs to outputs. Since we are interested in using FNNs for a specific task, we require this mapping to be optimal (keeping the hyperparameters and activation functions fixed whilst varying the weights and biases to achieve an optimal mapping). The concept of optimality is task specific and is not an inherent feature of the FNN. Optimality is characterised by a loss function

$l: \mathbb{R}^N \times \mathbb{R}^{\bar{N}} \to \mathbb{R}$ for some $\bar{N} \in \mathbb{N}$. Given an input $\boldsymbol{x} \in \mathbb{R}^I$ and a reference value $\boldsymbol{y} \in \mathbb{R}^{\bar{N}}$, the realised loss is defined as

$$l\left(\boldsymbol{f}\left(\boldsymbol{x}\right), \boldsymbol{y}\right).$$

Fixing the hyper-parameters and acitivation functions of the FNN

$$\boldsymbol{f} \in \mathcal{N}_r\left(I, d_1, \cdots, d_{r-1}, N; \boldsymbol{\sigma}_1, \cdots, \boldsymbol{\sigma}_r\right),$$

the network is parametrised by the vector $\boldsymbol{\theta}$ defined by

$$\boldsymbol{\theta} \coloneqq \left(\boldsymbol{W}^1, \cdots, \boldsymbol{W}^r; \boldsymbol{b}^1, \cdots, \boldsymbol{b}^r\right) \in \mathbb{R}^{d_1 \times d_0} \times \cdots \times \mathbb{R}^{d_r \times d_{r-1}} \times \mathbb{R}^{d_1} \times \cdots \times \mathbb{R}^{d_r}.$$

If $(\boldsymbol{x}, \boldsymbol{y})$ are realisations of some random vector $(\boldsymbol{X}, \boldsymbol{Y})$, one approach would be to find a FNN $\boldsymbol{f}^*$ such that $\boldsymbol{f}^*$ minimises risk, i.e.

$$\boldsymbol{f}^* = \min_{\boldsymbol{f}} \{\mathbb{E}\left(l\left(\boldsymbol{f}\left(\boldsymbol{X}\right), \boldsymbol{Y}\right)\right]\}.$$

However, in practice, the distribution of $(\boldsymbol{X}, \boldsymbol{Y})$ is not known and we need to use empirical methods with samples $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M \in \mathbb{R}^I$ and $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_M \in \mathbb{R}^N$ for some $M \in \mathbb{N}$. The optimal FNN $\boldsymbol{f}^*$ is constructed with the aim of reducing empirical risk, i.e.

$$\boldsymbol{f}^* = \min_{\boldsymbol{f}} \{\mathcal{L}\left(\boldsymbol{f}\right)\}, \quad \mathcal{L}\left(\boldsymbol{f}\right) \coloneqq \frac{1}{M} \sum_{i=1}^{M} l\left(\boldsymbol{f}\left(\boldsymbol{x}_i\right), \boldsymbol{y}_i\right). \tag{2.1}$$

Empirical risk over any subset $B \subseteq \{1, \cdots, M\}$ (B is called a minibatch), is referred to as minibatch risk and is defined by

$$\mathcal{L}_B\left(\boldsymbol{\theta}\right) \coloneqq \frac{1}{|B|} \sum_{i \in B} l\left(\boldsymbol{f}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\right), \boldsymbol{y}_i\right) \tag{2.2}$$

where $\boldsymbol{f}_{\boldsymbol{\theta}}$ denotes the FNN $\boldsymbol{f}$ with weights and biases given by $\boldsymbol{\theta}$.

Stochastic Gradient Descent

Since FNNs are trained by minimising empirical risk (Eq. (2.1)) we need a way of minimising this function. One of the methods used to minimise empirical risk is

stochastic gradient descent (SGD). In SGD, minibatches $B_1, \cdots, B_k \subseteq \{1, \cdots, M\}$ of fixed size $m << M, M = km$ for some $k \in \mathbb{N}$ are uniformly sampled without replacement, i.e.

$$\cup_{i=1}^{k} B_i = \{1, \cdots, M\},\ |B_i| = m,\ B_i \text{ are disjoint for } i = 1, \cdots k.$$

Starting from an initial parameter vector $\boldsymbol{\theta}_0$, the parameters are updated using the update scheme

$$\boldsymbol{\theta}_i \coloneqq \boldsymbol{\theta}_{i-1} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i} \left(\boldsymbol{\theta}_{i-1} \right),\ i = 1, \cdots, k. \tag{2.3}$$

One pass through the entire training data using the update scheme given in Eq. (2.3) constitutes an epoch. In SGD, this process is performed for a pre-defined number of epochs, each with its own (new) minibatches and initialising $\boldsymbol{\theta}$ with the last value of the previous epoch. η is a hyper-parameter which needs to be fine-tuned and is called the learning rate.

Adam

Adam (Algorithm 1) is another algorithm used to minimise empirical risk. The algorithm was introduced by Kingma and Ba [22]. It updates exponential moving averages of the gradient and the squared gradient. The moving averages correspond to the first moment (mean) and the second raw moment (uncentred variance) of the gradient respectively. The moving averages are initialised to vectors of all 0 and therefore the moment estimates are biased towards 0, especially during the initial updates. The decay rates β_0, β_1 are set close to 1. The moving averages $\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$ are the bias-corrected estimates. Through the second raw moment $\boldsymbol{v}$, Adam adapts itself to the loss function l via a diagonal Fisher matrix approximation $diag\left(\boldsymbol{v}\right)$ (Zhou et al. [44]). The step size $\left(\alpha \cdot \hat{\boldsymbol{m}}_i^e\right) / \left(\sqrt{\hat{\boldsymbol{v}}_i^e} + \epsilon\right)$ is a function of the bias-corrected moving averages.

SGD vs Adam and AdamW

To analyse the algorithmic performance of SGD and Adam, we formulate them as discretisations of stochastic differential equations (SDE). The objective function of

Algorithm 1: Adam. $(\alpha \cdot \hat{\boldsymbol{m}}_i^e)/\left(\sqrt{\hat{\boldsymbol{v}}_i^e}+\epsilon\right)$ indicates element-wise operations. Adam with L_2 regularization and Adam with decouple weight decay (AdamW) . SetScheduleMultiplier is a user defined function to account for scheduling of parameters .

Require: learning rate $\alpha > 0$
Require: decay rate of gradient moving average $\beta_0 \in [0, 1)$
Require: decay rate of squared gradient moving average $\beta_1 \in [0, 1)$
Require: initial parameter $\boldsymbol{\theta}_0$
Require: number of epochs E
Require: minibatch size $B \in \mathbb{N}$ such that $M = kB$ for some $k \in \mathbb{N}$
Input: ϵ
for $e = 1, \cdots, E$ **do**
  sample disjoint $B_1^e, \cdots, B_k^e \subset \{1, \cdots, M\}$ such that $\left|B_j^e\right| = B$ for $j = 1, \cdots, k$
  if $e = 1$ **then**
    $\boldsymbol{\theta}_0^e := \boldsymbol{\theta}_0$
    $\boldsymbol{m}_0^e = \mathbf{0}$
    $\boldsymbol{v}_0^e = \mathbf{0}$
  else
    $\boldsymbol{\theta}_0^e := \boldsymbol{\theta}_k^{e-1}$
    $\boldsymbol{m}_0^e = \boldsymbol{m}_k^{e-1}$
    $\boldsymbol{v}_0^e = \boldsymbol{v}_k^{e-1}$
  for $i = 1, \cdots, k$ **do**
    $\boldsymbol{g}_i^e = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i^e}\left(\boldsymbol{\theta}_{i-1}^e\right) + \lambda \boldsymbol{\theta}_{i-1}^e$ (gradients w.r.t parameters)
    $\boldsymbol{m}_i^e = \beta_1 \cdot \boldsymbol{m}_{i-1}^e + (1-\beta_1) \cdot \boldsymbol{g}_i^e$ (update biased first moment estimate)
    $\boldsymbol{v}_i^e = \beta_2 \cdot \boldsymbol{v}_{i-1}^e + (1-\beta_2) \cdot \left(\boldsymbol{g}_i^e \odot \boldsymbol{g}_i^e\right)$ (update biased second moment estimate)
    $\hat{\boldsymbol{m}}_i^e = \boldsymbol{m}_i^e / \left(1-\beta_1^{i+k(e-1)}\right)$ (bias-corrected first moment estimate)
    $\hat{\boldsymbol{v}}_i^e = \boldsymbol{v}_i^e / \left(1-\beta_2^{i+k(e-1)}\right)$ (bias-corrected second moment estimate)
    $\eta_i^e = SetScheduleMultiplier\left(i + k\left(e-1\right)\right)$ (scaling factor)
    $\boldsymbol{\theta}_i^e = \boldsymbol{\theta}_{i-1}^e - \eta_i^e \left((\alpha \cdot \hat{\boldsymbol{m}}_i^e)/\left(\sqrt{\hat{\boldsymbol{v}}_i^e}+\epsilon\right) + \lambda \boldsymbol{\theta}_{i-1}^e\right)$ (update parameters)

the neural network can be formulated as

$$\min_{\boldsymbol{\theta}} \boldsymbol{f}^*\left(\boldsymbol{\theta}\right) := \frac{1}{M}\sum_{i=1}^{M} f_i\left(\boldsymbol{\theta}\right)$$

where $f_i\left(\boldsymbol{\theta}\right)$ is the loss of the i^{th} sample, i.e.

$$f_i\left(\boldsymbol{\theta}\right) = l\left(\boldsymbol{f}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\right), \boldsymbol{y}_i\right).$$

Consider the minibatches $B_1, \cdots, B_k \subseteq \{1, \cdots, M\}$ of fixed size $m << M$, $M = km$ for some $k \in \mathbb{N}$. Gradient noise is defined by

$$\boldsymbol{u}_i := \nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*\left(\boldsymbol{\theta}_i\right) - \nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i}\left(\boldsymbol{\theta}_i\right)$$

where $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i}(\boldsymbol{\theta}_i) = \frac{1}{m} \sum_{\boldsymbol{x}_j \in B_i} \nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{x}_j; \boldsymbol{\theta}_i)$ (this is equivalent to the definition given in Eq. (2.2)). Substituting gradient noise into SGD's update rule, we get

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \eta \frac{1}{m} \sum_{\boldsymbol{x}_j \in B_i} \nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta}_i) = \boldsymbol{\theta}_{i-1} - \eta \nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*(\boldsymbol{\theta}_{i-1}) + \eta \boldsymbol{u}_i. \tag{2.4}$$

Definition 5. A random variable X is said to follow an $S\alpha S(\sigma)$-distribution iff the characteristic function of X, ϕ_X is given by

$$\phi_X(w) = \exp\left(-|\sigma\omega|^{\alpha}\right).$$

There is no closed-form formula for the density of a random variable following the $S\alpha S(\sigma)$-distribution, except for special cases, and the density decays with a power law tail like $1/|x|^{\alpha+1}$ where $\alpha \in (0, 2)$. α is called the tail index and determines the behaviour of the distribution since as α gets smaller the distribution has a heavier tail. $\sigma > 0$ is the scale parameter and controls the spread of X around 0.

In Şimşekli et al. [6] the gradient noise is modelled as an $S\alpha S(\boldsymbol{\Sigma})$-distributed random variable where $\boldsymbol{\Sigma}$ is an iteration-independent covariance matrix. On the other hand, Zhou et al. [44] model gradient noise as an $S\alpha S(\boldsymbol{\Sigma}_i)$-distributed random variable where $\boldsymbol{\Sigma}_i$ is an iteration-dependent covariance matrix. This is the approach we will use to compare the optimisation algorithms.

Definition 6. Let $(L^{\alpha})_{i \in \mathbb{N}}$ be a stochastic process. $(L^{\alpha})_{i \in \mathbb{N}}$ is said to be a Lévy Motion if

(i) $L_0^{\alpha} = 0$

(ii) For $t_0 < \cdots < t_N$ the increments $\left(L_{t_i}^{\alpha} - L_{t_{i-1}}^{\alpha}\right)$ are independent for $i = 1, \cdots, N$

(iii) For $s < t$, $(L_t^{\alpha} - L_s^{\alpha})$ and L_{t-s}^{α} have the same distribution: $S\alpha S\left((t-s)^{1/\alpha}\right)$

(iv) $(L_i^{\alpha})_{i \in \mathbb{N}}$ is continuous in probability.

Let $(\boldsymbol{L}_i)_{i \in \mathbb{N}}$ be a Lévy Motion such that $\boldsymbol{L}_i$ is a random vector where each of its components follow an $S\alpha S(1)$ distribution. The Lévy-driven SDE for SGD is given by

$$d\boldsymbol{\theta}_i = -\nabla_{\boldsymbol{\theta}} \boldsymbol{f}^* (\theta_i)\, di + \epsilon \boldsymbol{\Sigma}_i d\boldsymbol{L}_i$$

where $\epsilon \coloneqq \eta^{(\alpha-1)/\alpha}$. A derivation of this is provided in Appendix A.2.1.

Similarly, we derive the Lévy-driven SDE for Adam. Define

$$\boldsymbol{m}_i' \coloneqq \beta_1 \boldsymbol{m}_{i-1}' + (1-\beta_1) \nabla_{\boldsymbol{\theta}} \boldsymbol{f}^* (\boldsymbol{\theta}_i)$$

with $\boldsymbol{m}_0' = \boldsymbol{0}$. The Lévy-driven SDE for Adam is given by

$$d\boldsymbol{\theta}_i = -\mu_i \boldsymbol{Q}_i^{-1} \boldsymbol{m}_i' di + \epsilon \boldsymbol{Q}_i^{-1} \boldsymbol{\Sigma}_i d\boldsymbol{L}_i, \quad d\boldsymbol{m}_i' = \beta_1 \left(\nabla_{\boldsymbol{\theta}} \boldsymbol{f}^* (\boldsymbol{\theta}_i) - \boldsymbol{m}_i'\right) di,$$
$$d\boldsymbol{v}_i = \beta_2 \left[\nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i} (\boldsymbol{\theta}_i) \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i} (\boldsymbol{\theta}_i) - \boldsymbol{v}_i\right] di.$$

where $\epsilon = \eta^{(\alpha-1)/\alpha}$, $\boldsymbol{Q}_i = diag\left(\sqrt{w_i \boldsymbol{v}_i} + \epsilon\right)$, $u_i = 1/\left(1 - e^{-\beta_1 i}\right)$, and $w_i = 1/\left(1 - e^{-\beta_2 i}\right)$ are two constants which correct the bias in $\boldsymbol{m}_i'$ and $\boldsymbol{v}_i$, and $\odot$ is the component-wise Hadamard product. A detailed explanation of this SDE is provided in Appendix A.2.2.

Suppose the process $(\boldsymbol{\theta}_i)_{i \in \mathbb{N}}$ starts at a local basin Ω with minimum $\boldsymbol{\theta}^*$. Since it starts at Ω, $\boldsymbol{\theta}_0 \in \Omega$. Let $\partial\Omega$ denote the boundary of Ω and $\Omega^{-\epsilon^\gamma} \coloneqq \{\boldsymbol{y} \in \Omega \mid distance\left(\partial\Omega, \boldsymbol{y}\right) \geq \epsilon^\gamma\}$ denote the inner part of Ω. The escaping time Γ of the process $(\boldsymbol{\theta}_i)_{i \in \mathbb{N}}$ is defined as $\Gamma \coloneqq \inf\{i \geq 0 \mid \boldsymbol{\theta}_i \notin \Omega^{-\epsilon^\gamma}\}$ where the constant $\gamma > 0$ satisfies $\lim_{\epsilon \to 0} \epsilon^\gamma = 0$. Also, define the escaping set $\boldsymbol{W} \coloneqq \{\boldsymbol{y} \mid \boldsymbol{Q}_{\boldsymbol{\theta}^*}^{-1} \boldsymbol{\Sigma}_{\boldsymbol{\theta}^*} \boldsymbol{y} \notin \Omega^{-\epsilon^\gamma}\}$ where $\boldsymbol{\Sigma}_{\boldsymbol{\theta}^*} = \lim_{\boldsymbol{\theta}_i \to \boldsymbol{\theta}^*} \boldsymbol{\Sigma}_i$ for both SGD and Adam, $\boldsymbol{Q}_{\boldsymbol{\theta}^*} = \boldsymbol{I}$ for SGD, and $\boldsymbol{Q}_{\boldsymbol{\theta}^*} = \lim_{\boldsymbol{\theta}_i \to \boldsymbol{\theta}^*} \boldsymbol{Q}_i$ for Adam.

Definition 7. Let X be a Hausdorff topological space and $m(\cdot)$ be a measure on X. m is said to be a Radon measure if

(i) m is inner regular, i.e. $m(\mathcal{V}) = \sup_{\mathcal{U} \subseteq \mathcal{V}} m(\mathcal{U})$

(ii) m is outer regular, i.e. $m(\mathcal{V}) = \inf_{\mathcal{V} \subseteq \mathcal{U}} m(\mathcal{U})$

(iii) m is locally finite, i.e. every point of X has a neighbourhood $\mathcal{U}$ with finite

$m(\mathcal{U})$.

If in addition to conditions (i)-(iii) m obeys the condition $m(\mathcal{U}) < m(\mathcal{V})$ if $\mathcal{U} \subset \mathcal{V}$, m is said to be a non-zero Radon measure.

Under certain assumptions on the objective function (Zhou et al. [44, Section 4 *Analysis for Escaping Local Minima*, Assumption 1, page 5]) and assumptions on gradients of Adam (Zhou et al. [44, Section 4 *Analysis for Escaping Local Minima*, Assumption 2, page 5]), Zhou et al. [44, Section 4 *Analysis for Escaping Local Minima*, Theorem 1, page 6] show that when ϵ is small, for both SGD and Adam, the upper and lower bounds of their expected escaping time Γ are of the order of

$$\mathcal{O}\left(\frac{1}{m(\boldsymbol{W})\,\Theta(\epsilon^{-1})}\right) \tag{2.5}$$

where $\Theta(\epsilon^{-1}) := \frac{2}{\alpha}\epsilon^{\alpha}$ and $m(\boldsymbol{W})$ is a non-zero Radon measure on the escaping set (this value differs for SGD and Adam as they have different escaping sets). This result provides us with an indication of the performance of SGD and Adam. If the escaping time is large, the algorithm cannot easily escape the basin Ω and would get stuck in it. Moreover, given the same basin Ω, if an algorithm has a smaller escaping time than another, the algorithm with the smaller escaping time is more locally unstable and would escape the basin faster.

A minimum $\boldsymbol{\theta}^*$ is said to be flat if its basin has large non-zero Radon measure. SGD and Adam both have large escaping times at flat minima. Therefore, they would escape *sharp* minima because of their smaller escaping time and converge to flat ones. For a basin Ω, its Radon measure is proportional to its volume, however $m(\mathcal{W})$ is inversely proportional to the volume of Ω. Therefore, SGD and Adam are more stable at the minima with a larger basin in terms of volume. Intuitively, this can be interpreted that the volume of the basin determines the required jump size of the Lévy motion $(\boldsymbol{L}_t)_{t\in\mathbb{N}}$ in the SDEs to escape. Therefore, the larger the basin, the harder it is for the algorithm to escape.

Another difference between SGD and Adam is their generalisation performance. Keskar et al. [20] posit that minima at the flat or asymmetric basins often have

better generalisation performance on the test set. Since Eq. (2.5) provides clarity on the escape time of SGD and Adam from sharp minima to flat ones with larger Radon measure, we conclude that the faster the escaping time of the algorithm, the better its generalisation performance. Compared with SGD, the minima found by Adam often suffer from worse test performance (Keskar and Socher [21], Wilson et al. [40]). This generalisation difference can be explained through the gradient noise and geometry adaptation reflected by the factors $\Theta\left(\epsilon^{-1}\right)$ and $m\left(\mathcal{W}\right)$ respectively in Eq. (2.5). Since Adam considers exponential gradient noise $\frac{1-\beta_1}{1-\beta_1^i}\sum_{j=0}^{i}\beta_1^{i-j}\boldsymbol{u}_i$ whilst SGD assumes gradient noise to be $\boldsymbol{u}_i$, in Adam we smooth the gradient noise and therefore SGD leads to heavier tails of gradient noise. Thus, SGD has smaller tail index α for certain optimisation iterations which helps escaping behaviour. It follows that SGD is more locally unstable and converges to flat minima which are generally located at the flat or asymmetric basins and therefore benefits from better generalisation performance. An analysis of how the Radon measure of the escaping set helps in explaining the better generalisation performance of SGD is found in Zhou et al. [44, Section 4 *Analysis for Escaping Local Minima*, pages 6-8].

Loshchilov and Hutter [25] also investigate the generalisation discrepancies between SGD and Adam. To alleviate the generalisation issue, they propose Adam with decoupled weight decay (AdamW), an alternative version of Adam (Algorithm 1 blue section). SGD with weight decay is equivalent to SGD with L2 regularization (Loshchilov and Hutter [25, Section 2 *Decoupling The Weight Decay From The Gradient-Based Update*, Proposition 1, page 2]). However, this is not the case for Adam (Proposition 2 Loshchilov and Hutter [25, Section 2 *Decoupling The Weight Decay From The Gradient-Based Update*, Proposition 2, page 3]). In the case of Adam, using L2 regularization results in weights with large gradient magnitude being regularized by a smaller amount when compared with other weights. On the other hand, decoupled weight decay regularizes all weights with the same rate λ. This results in the regularization of weights with a large gradient (in terms of magnitude) being regularized more than with L2 regularization. Since Loshchilov and Hutter [25] show empirically that AdamW has better generalisation performance than Adam, the message classifier used for the purposes of this research is trained

with AdamW.

Backpropagation

To update the parameters, the gradient of minibatch empirical risk $\mathcal{L}_B$ needs to be computed numerically. A possible approach could be to approximate the gradient using finite differences, i.e.

$$\frac{\partial}{\partial \theta_i}\mathcal{L}_B\left(\boldsymbol{\theta}\right) \approx \frac{\mathcal{L}_B\left(\boldsymbol{\theta} + \frac{1}{2}\Delta \boldsymbol{e}_i\right) - \mathcal{L}_B\left(\boldsymbol{\theta} - \frac{1}{2}\Delta \boldsymbol{e}_i\right)}{\Delta}$$

where $\boldsymbol{e}_i$ is the all 0 vector with 1 in the i^{th} position and $\Delta > 0$.

If $\mathcal{L}_B$ is highly non-linear, this approximation yields poor results. Since FNNs are non-linear (the 'degree' of non-linearity generally increases as the number of hidden layers increase) finite difference methods are not well suited to approximate the gradient of minibatch empirical risk. Instead we use backpropagation, a special case of backward-mode algorithmic differentiation. Since

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}_B\left(\boldsymbol{\theta}\right) = \frac{1}{|B|}\sum_{i\in B}\nabla_{\boldsymbol{\theta}} l\left(\boldsymbol{f}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\right), \boldsymbol{y}_i\right)$$

we only need to compute the gradient of the loss function l for specific samples. Theorem 8 (Pakkanen [29]) formulates backpropagation and provides the formulae to calculate $\nabla_{\boldsymbol{\theta}} l$ for the specific case where all activation functions are the component-wise application of a one-dimensional function. Backpropagation can also be extended to cover activation functions which are not component-wise such as *softmax* which is defined by

$$softmax\colon \mathbb{R}^k \to [0,1]^k, \quad softmax\left(z_1, \cdots, z_k\right) = \left(\frac{e^{z_1}}{\sum_{j=1}^k e^{z_j}}, \cdots, \frac{e^{z_k}}{\sum_{j=1}^k e^{z_k}}\right).$$

Theorem 8. *Let* $\boldsymbol{f}_{\boldsymbol{\theta}} \in \mathcal{N}_r\left(I, d_1, \cdots, d_{r-1}, N; \boldsymbol{\sigma}_1, \cdots, \boldsymbol{\sigma}_r\right)$ *and* $\boldsymbol{x} \in \mathbb{R}^I$. *Suppose* $\boldsymbol{\sigma}_i$ *is the component-wise application of a one-dimensional function* $g_i\colon \mathbb{R} \to \mathbb{R}$ *and* $\boldsymbol{\sigma}_i'$ *is the component-wise derivative of* $\boldsymbol{\sigma}_i$, *i.e.* $\boldsymbol{\sigma}_i = \left(g_i, \cdots, g_i\right), \boldsymbol{\sigma}_i' = \left(g_i', \cdots, g_i'\right)$ *for*

$i = 1, \cdots, r$. *Define*

$$\begin{aligned}
&\boldsymbol{z}^i = \left(z_1^i, \cdots, z_{d_i}^i\right) \coloneqq \boldsymbol{L}^i\left(\boldsymbol{a}^{i-1}\right) = \boldsymbol{W}^i \boldsymbol{a}^{i-1} + \boldsymbol{b}^i, \qquad i = 1, \cdots, r \\
&\boldsymbol{a}^i = \left(a_1^i, \cdots, a_{d_i}^i\right) \coloneqq \boldsymbol{\sigma}_i\left(\boldsymbol{z}^i\right), \qquad\qquad\qquad i = 1, \cdots, r \\
&\boldsymbol{a}^0 = \boldsymbol{x}.
\end{aligned}$$

We also define the adjoint $\boldsymbol{\delta}^i = \left(\delta_1^i, \cdots, \delta_{d_i}^i\right) \in \mathbb{R}^{d_i}$ *by*

$$\delta_j^i \coloneqq \frac{\partial l}{\partial z_j^i}, \qquad j = 1, \cdots, d_i$$

for any $i = 1, \cdots, r$. *Then*

$$\boldsymbol{\delta}^r = \boldsymbol{\sigma}_r'\left(\boldsymbol{z}^r\right) \odot \nabla_{\hat{\boldsymbol{y}}} l\left(\boldsymbol{a}^r, \boldsymbol{y}\right) \tag{2.6}$$

$$\boldsymbol{\delta}^i = \boldsymbol{\sigma}_i'\left(\boldsymbol{z}^i\right) \odot \left(\boldsymbol{W}^{i+1}\right)^T \boldsymbol{\delta}^{i+1}, \qquad i = 1, \cdots, r-1 \tag{2.7}$$

$$\frac{\partial l}{\partial b_j^i} = \delta_j^i, \qquad i = 1, \cdots, r, \quad j = 1, \cdots, d_i \tag{2.8}$$

$$\frac{\partial l}{\partial W_{j,k}^i} = \delta_j^i a_k^{i-1}, \qquad i = 1, \cdots, r, \quad j = 1, \cdots, d_i, \quad k = 1, \cdots, d_{i-1} \tag{2.9}$$

where $\odot$ *is the component-wise Hadamard product of vectors.*

The proof of this result is provided in Appendix A.1.

In practice, to apply Theorem 8, $l\left(\boldsymbol{f}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right), \boldsymbol{y}\right)$ is first evaluated in a forward pass through the network

$$\boldsymbol{x} = \boldsymbol{a}^0 \to \boldsymbol{z}^1 \to \boldsymbol{a}^1 \to \cdots \to \boldsymbol{z}^r \to \boldsymbol{a}^r \to l\left(\boldsymbol{a}^r, \boldsymbol{y}\right) = l\left(\boldsymbol{f}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right), \boldsymbol{y}\right).$$

The intermediate values $\boldsymbol{a}^1, \cdots, \boldsymbol{a}^r$ and $\boldsymbol{z}^1, \cdots, \boldsymbol{z}^r$ are stored. Then backpropagation is performed using either symbolic or algorithmic differentiation of $g_1, \cdots, g_r$ and $\nabla_{\hat{\boldsymbol{y}}} l\left(\boldsymbol{a}^r, \boldsymbol{y}\right)$.

2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a collection of neural networks which incorporate memory into the models, thus making them more appropriate to handle sequential data. FNNs have proven to be very good models at handling a lot of tasks, however, their performance drops when dealing with sequential data. This is because they cannot feed the output back into the system (there is no concept of memory). For example, if trying to predict the next word in a sentence, being able to *remember* a substantial amount of the preceding words should produce a better output than if just the previous word is known. This is because more of the context is being considered when making the prediction. Another advantage of RNNs over FNNs when working with sequential data is that RNNs can work with data of variable length, this being a fundamental characteristic of sequential datasets (an example being that not all sentences have the same length). This is achieved through the concept of parameter sharing (described below). RNNs form the building block for LSTMs, a model featuring in the classification head of the message classifier used for this research. A detailed explanation of RNNs is provided in Goodfellow et al. [11, Chapter 10 *Sequence Modelling: Recurrent and Recursive Nets*, pages 367-415].

Consider a function $\boldsymbol{f}: \mathbb{R}^I \rightarrow \mathbb{R}^N$ which maps inputs of dimension I to outputs of dimension N and is parametrised by the vector $\boldsymbol{\theta}$. Consider the dynamic system

$$\boldsymbol{h}^t = \boldsymbol{f}\left(\boldsymbol{h}^{t-1}; \boldsymbol{\theta}\right).$$

This is a recurrent system and can be unfolded as follows

$$\boldsymbol{h}^t = \boldsymbol{f}\left(\boldsymbol{f}\left(\boldsymbol{h}^{t-2}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right).$$

The unfolding can take place until the initial state is reached. Through unfolding, we demonstrate a fundamental principle of RNNs; parameter sharing. The parameter $\boldsymbol{\theta}$ is passed down through the system. This allows the system to handle inputs of variable length. This is essential for handling sequential data since, once the model

is deployed, unseen sequence lengths will be common.

In most RNNs, besides passing forward the value of the previous state, a new input is also inserted into the system. To account for this, we change the definition of $\boldsymbol{f}$ to $\boldsymbol{f}\colon \mathbb{R}^N \times \mathbb{R}^I \to \mathbb{R}^N$. Let $\boldsymbol{h}^t, \boldsymbol{x}^t$ be the output and input at sequence step t respectively. For $t \geq 1$,

$$\boldsymbol{h}^t = \boldsymbol{f}\left(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t; \boldsymbol{\theta}\right)$$

and $\boldsymbol{h}^0$ is the initial state.

For each step $t = 1, \cdots, T$ the following update equations are applied

$$\begin{aligned}
\boldsymbol{a}^t &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{t-1} + \boldsymbol{U}\boldsymbol{x}^t \\
\boldsymbol{h}^t &= \boldsymbol{\sigma}_1\left(\boldsymbol{a}^t\right) \\
\boldsymbol{o}^t &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^t \\
\boldsymbol{y}^t &= \boldsymbol{\sigma}_2\left(\boldsymbol{o}^t\right)
\end{aligned}$$

where $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2$ are activation functions, $\boldsymbol{b}, \boldsymbol{c}$ are bias vectors, and $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{V}$ are weight matrices. The weight matrices and bias vectors are shared amongst the unfolded layers in the neural network. Running through all the elements of a sequence constitutes a forward propagation. A common architecture of a RNN is depicted in Figure 2.2.

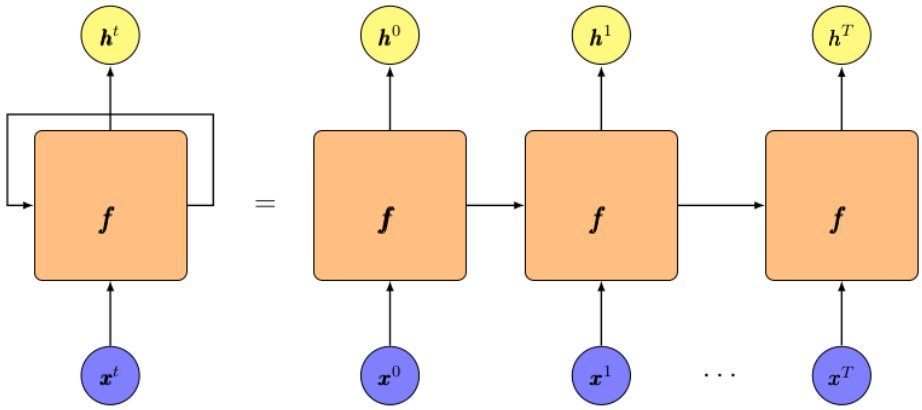


Figure 2.2: Common structure of RNN. The right-hand side is the unrolled version of the left-hand side.

2.2.1 Training RNNs

Training RNNs is similar to training FNNs. They both use gradient-based techniques. However, instead of backpropagation, RNNs use the more general technique of backpropagation through time (BPTT) (Chen [5]). Suppose we need to minimise the loss function

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}^t$$

where $\mathcal{L}^t$ is the loss associated with the label $\boldsymbol{y}^t$ (loss of element t in the sequence). Let $\hat{\boldsymbol{y}}^t$ denote the predicted label at time t ($\hat{\boldsymbol{y}}^t = \boldsymbol{\sigma}_2(\boldsymbol{o}^t)$). To calculate the partial derivative of $\mathcal{L}$ with respect to the matrix $\boldsymbol{W}$, we first need the partial derivative of the loss at time t with respect to $\boldsymbol{W}$. At time step $t \to t+1$, applying the chain rule, we get

$$\frac{\partial \mathcal{L}^{t+1}}{\partial \boldsymbol{W}} := \frac{\partial \mathcal{L}^{t+1}}{\partial \hat{\boldsymbol{y}}^{t+1}} \frac{\partial \hat{\boldsymbol{y}}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{W}}.$$

Since $\boldsymbol{W}$ is shared across all recursive units, $\boldsymbol{h}^{t+1}$ has a direct dependence on $\boldsymbol{W}$ and an implicit dependence through $\boldsymbol{h}^t$. Applying BPTT, it follows that

$$\frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{W}} \to \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{W}} + \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^t} \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{W}}.$$

Repeating the same process for $\partial \boldsymbol{h}^t / \partial \boldsymbol{W}$, substituting, and noting that

$$\frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^t} \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t-1}} \cdots \frac{\partial \boldsymbol{h}^j}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^j} \frac{\partial \boldsymbol{h}^j}{\partial \boldsymbol{W}}$$

for $j = 1, \cdots, t-1$, we obtain

$$\frac{\partial \mathcal{L}^{t+1}}{\partial \boldsymbol{W}} = \frac{\partial \mathcal{L}^{t+1}}{\partial \hat{\boldsymbol{y}}^{t+1}} \frac{\partial \hat{\boldsymbol{y}}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{t+1}}{\partial \hat{\boldsymbol{y}}^{t+1}} \frac{\partial \hat{\boldsymbol{y}}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^k} \frac{\partial \boldsymbol{h}^k}{\partial \boldsymbol{W}}.$$

Then

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \sum_{t=0}^{T-1} \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{t+1}}{\partial \hat{\boldsymbol{y}}^{t+1}} \frac{\partial \hat{\boldsymbol{y}}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{W}} = \sum_{t=0}^{T-1} \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{t+1}}{\partial \hat{\boldsymbol{y}}^{t+1}} \frac{\partial \hat{\boldsymbol{y}}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^k} \frac{\partial \boldsymbol{h}^k}{\partial \boldsymbol{W}}.$$

Similarly, applying BPTT,

$$\frac{\partial \mathcal{L}^{t+1}}{\partial \boldsymbol{U}} = \frac{\partial \mathcal{L}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{U}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^{k}} \frac{\partial \boldsymbol{h}^{k}}{\partial \boldsymbol{U}}.$$

Summing up the derivative at all time steps we get

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{U}} = \sum_{t=0}^{T-1} \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{t+1}}{\partial \boldsymbol{h}^{t+1}} \frac{\partial \boldsymbol{h}^{t+1}}{\partial \boldsymbol{h}^{k}} \frac{\partial \boldsymbol{h}^{k}}{\partial \boldsymbol{U}}.$$

Also,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{t}}{\partial \hat{\boldsymbol{y}}^{t}} \frac{\partial \hat{\boldsymbol{y}}^{t}}{\partial \boldsymbol{V}}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{t}}{\partial \hat{\boldsymbol{y}}^{t}} \frac{\partial \hat{\boldsymbol{y}}^{t}}{\partial \boldsymbol{c}}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{t}}{\partial \hat{\boldsymbol{y}}^{t}} \frac{\partial \hat{\boldsymbol{y}}^{t}}{\partial \boldsymbol{h}^{t}} \frac{\partial \boldsymbol{h}^{t}}{\partial \boldsymbol{b}}.$$

The parameters of the RNN are updated using a parameter update scheme such as AdamW.

2.2.2 Vanishing and Exploding Gradient Problem

The main idea underpinning RNNs is the need to model long-term dependencies in sequential data. In the optimisation process, this causes a major issue; gradients propagated over many stages tend to either vanish or explode (the former being more common than the latter). To demonstrate this problem, consider the simple RNN $\boldsymbol{h}^t = \boldsymbol{W}^T \boldsymbol{h}^{t-1}$ which lacks a non-linear activation function and input $\boldsymbol{x}$. The RNN simplifies to $\boldsymbol{h}^t = \left(\boldsymbol{W}^t\right)^T \boldsymbol{h}^0$. If $\boldsymbol{W}$ admits the eigendecomposition $\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$ where $\boldsymbol{Q}$ is an orthogonal matrix, the recurrence relation simplifies to $\boldsymbol{h}^t = \boldsymbol{Q}^T \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^0$. The eigenvalues are raised to the power of t, and therefore if the eigenvalue has magnitude less than 1 it decays to 0, otherwise it explodes. Any element of $\boldsymbol{h}^0$ that is not aligned with the largest (in terms of absolute value) eigenvalue is discarded.

2.2.3 Bi-directional RNN

The RNNs described so far condition on the previous values in the sequence. If the future values in the sequence are known in advance (such as text classification), they provide useful additional information which should be exploited by the model. Irsoy and Cardie [19] present a bi-directional deep RNN. At each position in the sequence

t, the network maintains two hidden layers, one for the left-to-right direction and another for the right-to-left direction. Handling two hidden states comes with the drawback of consuming double the memory required to store the weight matrices and bias vectors than a regular RNN would. The final output is a concatenation of both RNN hidden layers. This is given by the following equations

$$\overrightarrow{\boldsymbol{h}^t} = \boldsymbol{\sigma}_1\left(\overrightarrow{\boldsymbol{W}}\overrightarrow{\boldsymbol{h}^{t-1}} + \overrightarrow{\boldsymbol{U}}\boldsymbol{x}^t + \overrightarrow{\boldsymbol{b}}\right)$$
$$\overleftarrow{\boldsymbol{h}^t} = \boldsymbol{\sigma}_1\left(\overleftarrow{\boldsymbol{W}}\overleftarrow{\boldsymbol{h}^{t-1}} + \overleftarrow{\boldsymbol{U}}\boldsymbol{x}^t + \overleftarrow{\boldsymbol{b}}\right)$$
$$\hat{\boldsymbol{y}^t} = \boldsymbol{\sigma}_2\left(\boldsymbol{W}\left[\overrightarrow{\boldsymbol{h}^t}; \overleftarrow{\boldsymbol{h}^t}\right] + \boldsymbol{c}\right)$$

where $\left[\overrightarrow{\boldsymbol{h}^t}; \overleftarrow{\boldsymbol{h}^t}\right]$ is the concatenation of the two vectors.

2.2.4 Long Short-Term Memory Model

As shown in Bengio et al. [3], empirically RNNs have difficulty in capturing long-term dependencies in sequential data. A reason why this occurs is because of the vanishing gradient problem. A solution to this was proposed by Hochreiter and Schmidhuber [17] where they introduced the LSTM model. Since LSTMs are better than RNNs at capturing long-term dependencies, the message classifier adopted in this research incorporates an LSTM as part of the classification head.

The RNNs described above consist of an unfolded neural network layer, where by neural network layer we mean the composition of an activation function with an affine function. LSTMs are composed of four neural network layers, each performing a specific function. An LSTM layer is provided in Figure 2.3 (Olah [28]). The red nodes represent element-wise operations, the green nodes represent a neural network layer with activation function $\boldsymbol{\sigma}^*$ (sigmoid function) or **tanh**, and the arrow from input $\boldsymbol{x}^t$ merging with the line of $\boldsymbol{h}^{t-1}$ symbolises that these are the two inputs to the neural network layers. In an LSTM network, the LSTM layers will be connected to form a chain like structure. The sigmoid and **tanh** activation functions

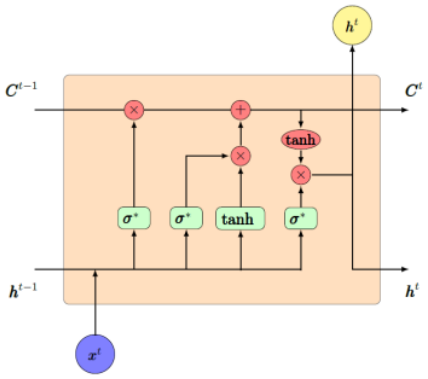


Figure 2.3: LSTM layer.

are defined as

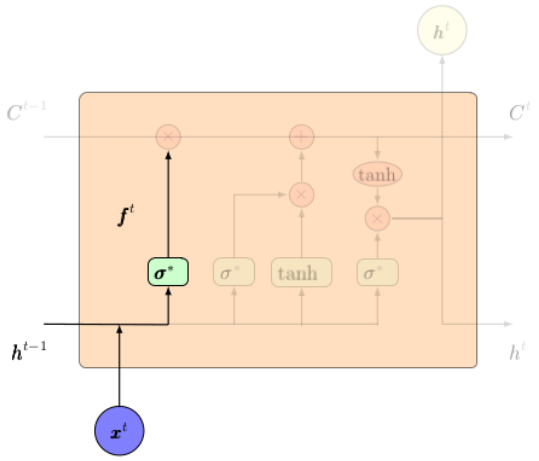
$$\boldsymbol{\sigma}^*: \mathbb{R}^n \to \mathbb{R}^n, \quad \boldsymbol{\sigma}^*\left(x_1, \cdots, x_n\right) \coloneqq \left(\frac{1}{1+e^{-x_1}}, \cdots, \frac{1}{1+e^{-x_n}}\right)$$

$$\textbf{tanh}: \mathbb{R}^n \to \mathbb{R}^n, \quad \textbf{tanh}\left(x_1, \cdots, x_n\right) \coloneqq \left(\frac{e^{x_1}-e^{-x_1}}{e^{x_1}+e^{-x_1}}, \cdots, \frac{e^{x_n}-e^{-x_n}}{e^{x_n}+e^{-x_n}}\right).$$

The cell state ($\boldsymbol{C}$) runs straight through the entire network, where information flows along it mostly unchanged. Gates are a way of optionally letting additional information into the cell state. The gates are composed of a neural network layer with a sigmoid activation function. The sigmoid function outputs values in the range $[0, 1]$, where the number signifies how much information to let through. A value of 0 translates to *no new information into the cell state* and a value of 1 means *let all information through.*

Forget Gate

The forget gate (Figure 2.4) is used to determine which information of the cell state to *forget* and therefore which information to keep. This decision is made by a neural network layer which takes as input the previous state's value $\boldsymbol{h}^{t-1}$ and the current input $\boldsymbol{x}^t$. As previously mentioned, the sigmoid function outputs values from 0 to 1 (inclusive of both) and a value of 0 means to *forget* completely and a 1 represents to *remember* fully. The forget function $\boldsymbol{f}^t$ is governed by the equation in Figure 2.4.

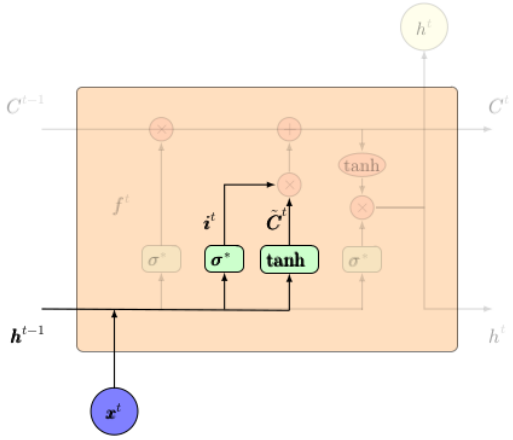


$$f^t = \sigma^* \left(W^f h^{t-1} + U^f x^t + b^f \right)$$

Figure 2.4: Forget Gate.

Input Gate and New Memory Generation

The new memory $\tilde{C}^t$ is generated using the new input x^t and past hidden state h^{t-1}. However, the new memory generation stage does not check the relevance of the new input to the overall objective before generating the new memory. This is performed by the input gate. The input gate uses the input x^t and the past hidden state h^{t-1} to determine whether the new memory $\tilde{C}^t$ is worth preserving. i^t contains values in the range $[0, 1]$ where a 0 in the j^{th} position means discard the j^{th} part of the new memory and a value of 1 means to completely preserve it. The function definitions of $\tilde{C}^t$ and i^t are provided in Figure 2.5



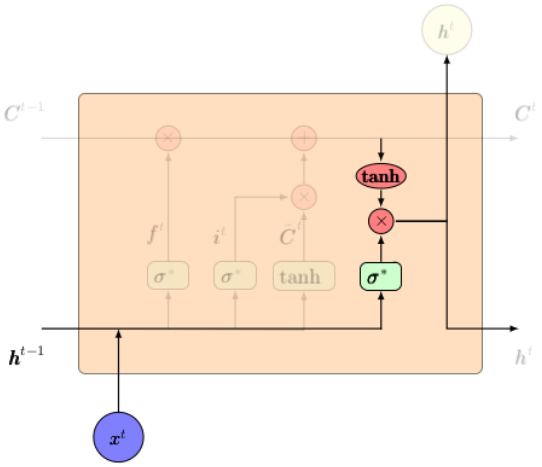
$$i^t = \sigma^* \left(W^i h^{t-1} + U^i x^t + b^i \right)$$

$$\tilde{C}^t = \tanh \left(W^C h^{t-1} + U^C x^t + b^C \right)$$

Figure 2.5: Input Gate and New Memory Generation.

Output Gate

The purpose of the output gate is to separate the final memory from the hidden state. This is important because the final memory $\boldsymbol{C}^t$ contains a lot more information than is necessary to be saved in the hidden state $\boldsymbol{h}^t$. This determines which part of the memory should propagate into the hidden state. The signal which encodes which parts are relevant is $\boldsymbol{o}^t$. Point-wise tanh is then performed with the memory. The architecture and relevant equations are provided in Figure 2.6.



$$\boldsymbol{o}^t = \boldsymbol{\sigma}^* \left(\boldsymbol{W}^o \boldsymbol{h}^{t-1} + \boldsymbol{U}^o \boldsymbol{x}^t + \boldsymbol{b}^o \right)$$

$$\boldsymbol{h}^t = \boldsymbol{o}^t \times \tanh \left(\boldsymbol{C}^t \right)$$

Figure 2.6: Output Gate.

2.3 Transformer

The Transformer is a model architecture introduced in Vaswani et al. [38]. The Transformer relies fully on an attention mechanism as opposed to recurrence to formulate dependencies between input and output. It consists of an encoder-decoder structure. The encoder maps an input sequence of symbol representations $(x_1, \cdots, x_n)$ to a sequence of continuous representations $\boldsymbol{z} = (z_1, \cdots, z_n)$. $\boldsymbol{z}$ is then fed into the decoder which generates an output sequence $(y_1, \cdots, y_m)$ of symbols, one element at a time. At each step, the model consumes the already generated symbols as additional input when generating the next output in the sequence. The message classifier uses the encoder structure to construct the text embeddings. The architecture of the model is depicted in Figure 2.7. We now describe each component of

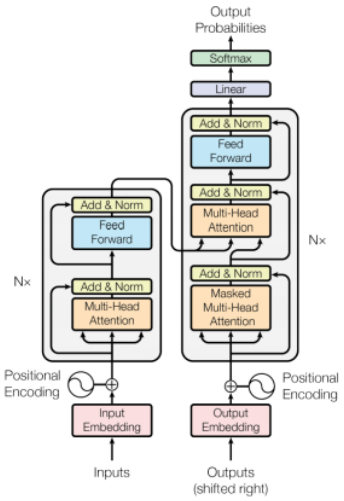


Figure 2.7: Transformer architecture. The encoder is on the left half and the decoder is on the right half (Vaswani et al. [38, Section 3 *Model Architecture*, page 3]).

the Transformer in detail.

2.3.1 Residual Connection

Inside each layer of the encoder and decoder, a residual connection is used around each sub-layer. Residual connections were introduced by He et al. [15] in their formulation of a deep residual learning framework. The motivation behind this framework is addressing the degradation problem; as network depth increases, the accuracy gets saturated and then degrades rapidly. This problem was not caused by overfitting, and adding more layers increased to higher training errors as was reported in He et al. [15] and He and Sun [14]. This problem should not occur. If we consider a shallow architecture and its deeper counterpart that adds more layers onto it, the deeper model should produce a training error which is not higher than the shallower one since by adding layers to the shallow network which consist of identity mappings, we can construct the deeper network.

To overcome this problem, the layers are constructed to fit a residual mapping. Formally, using the notation of He et al. [15], the stacked layers fit the residual mapping $\boldsymbol{F}(\boldsymbol{x}) \coloneqq \boldsymbol{F}^*(\boldsymbol{x}) - \boldsymbol{x}$ where $\boldsymbol{F}^*(x)$ is the desired underlying mapping. Then

$\boldsymbol{F}^{*}(\boldsymbol{x})$ is recast into $\boldsymbol{F}(\boldsymbol{x})+\boldsymbol{x}$. This is depicted in Figure 2.8.

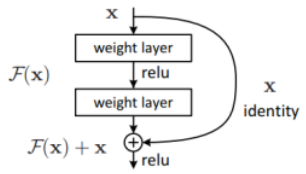


Figure 2.8: Residual mapping building block architecture (He et al. [15, Section 1 *Introduction*, page 771]).

As previously mentioned, a deeper network can be constructed from a shallower network by adding identity layers on top of the shallower network. Since the degradation problem persists, the solvers might be encountering difficulties when trying to approximate identity mappings. By working with the residual, if the identity mappings are optimal, the solvers can drive the weights of the additional layers to zero. In this way we will still be constructing an identity mapping. This is the motivation behind using residual connections.

2.3.2 Normalisation

Internal Covariate Shift

A learning system is said to experience covariate shift when the input distribution to the system changes (Shimodaira [34]). In Ioffe and Szegedy [18], covariate shift is extended to cover not only the whole system but the internal sub-layers of the system. In the case of neural networks, internal covariate shift extends covariate shift to include the sub-layers of the network. Consider the network

$$\boldsymbol{f}=\boldsymbol{F}_2\left(\boldsymbol{F}_1\left(\boldsymbol{x},\boldsymbol{\theta}_1\right),\boldsymbol{\theta}_2\right)$$

where $\boldsymbol{F}_1,\boldsymbol{F}_2$ are arbitrary transformations and parameters $\boldsymbol{\theta}_1,\boldsymbol{\theta}_2$ are to be learned by minimising a loss function l. Learning the parameters $\boldsymbol{\theta}_2$ can be treated as if the inputs to the system $\boldsymbol{F}_2\left(\cdot,\boldsymbol{\theta}_2\right)$ are $\boldsymbol{z}=\boldsymbol{F}_1\left(\boldsymbol{x},\boldsymbol{\theta}_1\right)$, i.e.

$$\boldsymbol{f}=\boldsymbol{F}_2\left(\boldsymbol{z},\boldsymbol{\theta}_2\right).$$

Using this representation, a gradient update step is equivalent to that of a stand-alone network with input $\boldsymbol{z}$. Therefore, properties of the input distribution that make training a network more efficient also apply to the internal sub-layers of the network. In particular, it is advantageous to keep $\boldsymbol{z}$'s distribution fixed so $\boldsymbol{\theta}_2$ would not need to readjust to distributional changes in $\boldsymbol{z}$.

Layer Normalisation

Normalisation is used to improve the training speed and generalisation capability of a neural network. Layer normalisation is described in Ba et al. [1]. Consider the l^{th} hidden layer of a FNN and let $\boldsymbol{a}^{l-1} \in \mathbb{R}^{d_{l-1}}$ be the output of the $(l-1)^{\text{th}}$ layer. Define

$$\boldsymbol{q}^l \coloneqq \boldsymbol{W}^l \boldsymbol{a}^{l-1} = \left(q_1^l, \cdots, q_{d_l}^l\right) \in \mathbb{R}^{d_l}.$$

Changes in $\boldsymbol{a}^{l-1}$ will cause highly correlated changes in $\boldsymbol{q}^l$. To try and remedy the internal covariance shift problem, the mean and variance of $\boldsymbol{a}^l$ can be fixed for each layer l. Define

$$\mu^l \coloneqq d_l^{-1} \sum_{i=1}^{d_l} q_i^l \quad \text{and} \quad \sigma^l \coloneqq \sqrt{d_l^{-1} \sum_{i=1}^{d_l} \left(q_i^l - \mu^l\right)^2}.$$

Then we define

$$\overline{q^l}_i \coloneqq \frac{g_i^l}{\sigma^l} \left(q_i^l - \mu^l\right)$$

where g_i is a gain parameter scaling the normalized activation before the non-linear activation function. The output of the l^{th} layer is given by

$$\boldsymbol{a}^l = \boldsymbol{\sigma}^l \left(\overline{\boldsymbol{q}^l} + \boldsymbol{b}^l\right)$$

where $\boldsymbol{\sigma}^l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_l}$ is the activation function, $\boldsymbol{b}^l \in \mathbb{R}^{d_l}$ is the bias and $\overline{\boldsymbol{q}^l} \coloneqq \left(\overline{q^l}_i, \cdots, \overline{q^l}_{d_l}\right) \in \mathbb{R}^{d_l}$. In practice, given any vector $\boldsymbol{q}^l$, the output of layer normalisation is given by

$$LN_{\gamma,\beta}\left(\boldsymbol{q}^l\right) \coloneqq \gamma \frac{\boldsymbol{q}^l - \mu^l}{\sqrt{\left(\sigma^l\right)^2 + \epsilon}} + \beta$$

where $\gamma, \beta \in \mathbb{R}$ are learnable parameters, $\epsilon \in \mathbb{R}$ is a hyper-parameter, and the multiplication and addition in the definition of $LN_{\gamma,\beta}$ are performed component-wise.

Layer normalisation is invariant to scaling and shifting of the entire weight matrix $\boldsymbol{W}^l$ and the input data. These properties of layer normalisation are articulated in Theorems 9 and 10.

Theorem 9. *Suppose* $\boldsymbol{\theta}_1^l = \left(\boldsymbol{W}_1^l, \boldsymbol{b}^l\right), \boldsymbol{\theta}_2^l = \left(\boldsymbol{W}_2^l, \boldsymbol{b}^l\right)$ *are two model parameters such that the weight matrices* $\boldsymbol{W}_1^l$ *and* $\boldsymbol{W}_2^l$ *differ by the scaling factor* δ *and the weights in* $\boldsymbol{W}_2^l$ *are shifted by a constant vector* $\boldsymbol{\gamma} \in \mathbb{R}^{d_{l-1}}$*, i.e.* $\boldsymbol{W}_2^l = \delta \boldsymbol{W}_1^l + \mathbf{1}_{d_l} \boldsymbol{\gamma}^T$ *where* $\mathbf{1}_{d_l} \in \mathbb{R}^{d_l}$ *is the all* 1*-column vector and* $\boldsymbol{\gamma}^T$ *denotes the transpose of* $\boldsymbol{\gamma}$*. Let* $\boldsymbol{a}_i^l$ *denote the output using parameter* $\boldsymbol{\theta}_i$ *for* $i = 1, 2$*. Then* $\boldsymbol{a}_2^l = \boldsymbol{a}_1^l$*.*

A proof of this result is provided in Appendix A.3.

Theorem 10. *Suppose* $\boldsymbol{a}_1^{l-1}, \boldsymbol{a}_2^{l-1}$ *are two data instances such that* $\boldsymbol{a}_2^{l-1} = \delta \boldsymbol{a}_1^{l-1} + \gamma \mathbf{1}_{d_{l-1}}$ *where* $\gamma, \delta \in \mathbb{R}$*. Let* $\boldsymbol{a}_i^l$ *denote the output corresponding to input* $\boldsymbol{a}_i^{l-1}$ *for* $i = 1, 2$*. Then* $\boldsymbol{a}_1^l = \boldsymbol{a}_2^l$*.*

Proof. The proof is very similar to the one provided for Theorem 9. □

2.3.3 Encoder

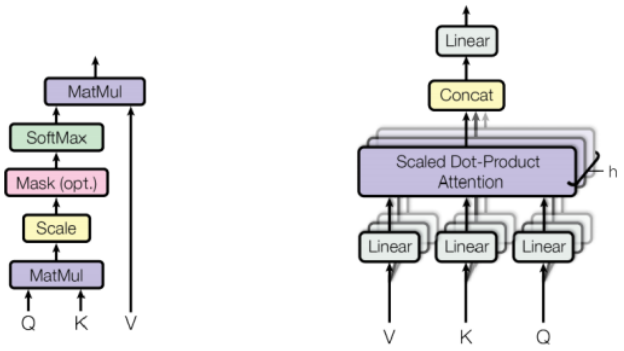
The encoder (left half of Figure 2.7) comprises of N stacked identical layers. Each layer consists of two sub-layers. The first sub-layer is a multi-head self-attention mechanism and the second sub-layer consists of a fully connected FNN.

2.3.4 Attention

An attention function is a mapping from a query and a set of key-value pairs to an output. The output is a weighted sum of the values and the weight assigned to each value is the result of a compatibility function of the query with the corresponding key. The query, key-value pairs and output are all vectors.

Scaled Dot-Product Attention

The scaled dot-product attention model is depicted in Figure 2.9a. The inputs are the query and key vectors of dimension d_k and the value vector of dimension d_v. The



(a) Scaled Dot-Product Attention. (b) Multi-Head Attention.

Figure 2.9: Attention Mechanism (Vaswani et al. [38, Section 3 *Model Architecture*, page 4]).

dot-product of the query with all the keys is computed. They are then normalized by dividing by $\sqrt{d_k}$ and the *softmax* function is applied to obtain the weights on the values. An optional mask is included before the *softmax* function is applied. In practice, the attention function on a set of queries is computed simultaneously where the queries are contained in the matrix $\boldsymbol{Q}$. The keys and values are also packed together into the matrices $\boldsymbol{K}$ and $\boldsymbol{V}$ respectively. The matrix of outputs is computed using the following formula

$$Attention\left(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}\right) := softmax\left(\frac{1}{\sqrt{d_k}}\boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V}.$$

Attention is used to help the model better understand dependencies in the text. For example, given the input sentence

"BoE discussed quantitative easing. They also mentioned interest rates.",

attention allows the model to associate "They" with "BoE". As each word is processed by the model, *attention* looks at other words in the input text to determine a better encoding for this word.

Multi-Head Attention

To capture information from different representation subspaces at different positions, the queries, keys, and values are linearly projected h times with different, learned linear projections to d_k, d_k, and d_v dimensions respectively. On each of these pro-

jected queries, keys, and values scaled dot-product attention is performed in parallel. This results in d_v-dimensional output values. These are concatenated and once again projected to yield the final attention scores. The structure of multi-head attention is depicted in Figure 2.9b. Let $d_{model} \in \mathbb{N}$ denote the output dimension of all sub-layers in the model as well as embeddings. Then

$$Multihead\left(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}\right) \coloneqq Concat\left(head_1, \cdots, head_h\right) \boldsymbol{W}^O$$

where

$$head_i \coloneqq Attention\left(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V\right)$$

$\boldsymbol{W}_i^Q, \boldsymbol{W}_i^K \in \mathbb{R}^{d_{model} \times d_k}, \boldsymbol{W}_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d_{model}}$.

A more detailed explanation of multi-head attention (similar to the one presented in Doshi [9]) is provided in Appendix C.

2.3.5 Position-wise FNN

Each layer of the encoder and decoder contains a fully connected FNN. The FNN is applied to each position separately and identically. One of the most widespread activation functions is the Rectified Linear Unit (ReLU) activation function defined as

$$ReLU\colon \mathbb{R} \to \mathbb{R}^+ \cup \{0\}, \quad ReLU\left(x\right) \coloneqq max\left(0, x\right) = x^+$$

where $\mathbb{R}^+$ denotes the positive real numbers. The FNN consists of one hidden layer and ReLU activation function in between, i.e.

$$\boldsymbol{f} \in \mathcal{N}_2\left(d_{model}, d_f, d_{model}; ReLU, Identity\right), \quad \boldsymbol{f}\left(\boldsymbol{x}\right) = \boldsymbol{W}^2 \max\left(0, \boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1\right) + \boldsymbol{b}^2$$

where d_f is the dimensionality of the inner layer.

2.3.6 Positional Encoding

Since the Transformer does not use recurrence or convolution, to make use of the order of the sequence, information about the relative or absolute position of the words

must be injected into the model. This is achieved by adding positional encodings to the input embeddings before we start the encoding and the decoding process. So that the positional encodings can be added to the embedding, they are of dimension d_{model}. The two main types of positional encodings are learned and fixed.

A common type of fixed positional encodings consists of using sine and cosine functions of different frequencies

$$PE_{(pos,2i)} \coloneqq \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \quad PE_{(pos,2i+1)} \coloneqq \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where pos is the position and i is the dimension. Each dimension of the positional encoding corresponds to a sinusoid and the wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. This encoding should allow the model to account for relative positions because for any fixed offset k, PE_{pos+k} can be formulated as a linear function of PE_{pos}.

In learned positional embeddings, the embedding is the output of an embedding layer. The parameters of the embedding layer are optimised during model training.

2.3.7 Decoder

The decoder consists of N identical stacked layers. Each layer consists of a masked multi-head attention layer, a multi-head attention layer, and a FNN.

2.4 BERT

When learning something new, we generally apply our pre-existing knowledge to help in better understanding the new material. This is the general idea behind pre-training. In pre-training a model is trained using a general method (not task specific) and is then fine-tuned to the downstream task. In terms of parameters, this translates to parameters being learnt to fit generalised tasks and then fine-tuned to fit the specific task. Language pre-training has emerged as an effective technique for improving many NLP tasks (Dai and Le [7] and Peters et al. [31]).

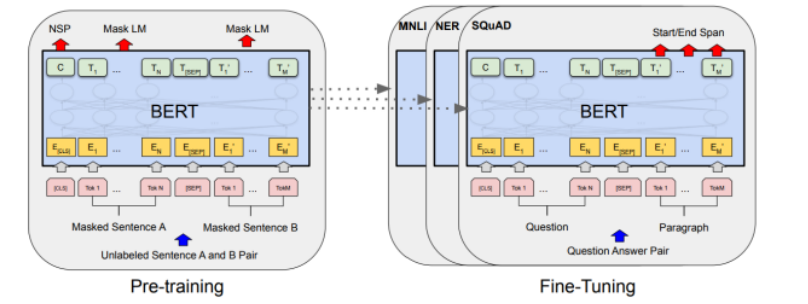


Figure 2.10: BERT pre-training and fine-tuning procedures (Devlin et al. [8, Section 3 *Bert*, Figure 1, page 3]).

BERT (Devlin et al. [8], Figure 2.10) is a pre-trained bi-directional encoder. Bi-directionality means that both the left-to-right context as well as the right-to-left context of sentences are taken into account when training the encoder. Bi-directionality is achieved through the pre-training methods used. BERT's architecture mainly comprises of the Transformer's encoder structure. We denote the number of stacked layers by L, the hidden size by H, and the number of self-attention heads by A. The message classifier used for this research constructs the encodings using BERT.

2.4.1 Input Representations

Throughout this section, a *sentence* refers to an arbitrary sequence of contiguous text and not a linguistic sentence. Given a sentence, tokenization is the process of breaking up the sentence into pieces called tokens. For example, the sentence "BoE discussed quantitative easing. They also mentioned interest rates" can be tokenized into the following tokens

| BoE | discussed | quantitative | easing | they | also | mentioned | interest | rates |

For BERT to be able to handle most downstream tasks such as classification and language translation, the input sequence must unambiguously cater for both single sentences and pairs of sentences. In our case, *sequence* refers to the input token

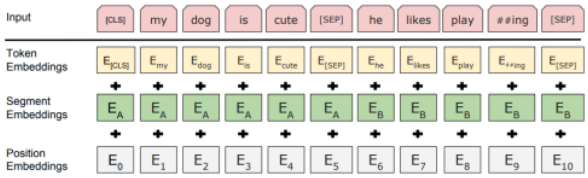


Figure 2.11: BERT input embeddings are the sum of the token embeddings, the segmentation embeddings, and the positional embeddings (Devlin et al. [8, Section 3 *Bert*, Figure 2, page 5]).

sequence to BERT.

The first token of every sequence is the special classification token *[CLS]*. The final hidden representation of the *[CLS]* token is used as the input vector for classification. Wordpiece embeddings are used to perform tokenization on the input sentence (Wu et al. [41]). Given a corpus of tokens, a greedy algorithm is used to break up the text into the longest-match-first tokens. The characters "##" are added to the beginning of the tokens for words which are split, besides for the first token. For example, if "playing" is split into the tokens "play" and "ing", the tokens will be represented as "play" and "##ing". The algorithm is provided in Appendix B (Algorithm 2). Sentence pairs are separated using the special token *[SEP]*. A learned segmentation embedding is added to every token to indicate whether it is the first (sentence A) or second (sentence B) sentence. The input embeddings are the sum of the token embeddings, the segmentation embeddings, and the position embeddings (Figure 2.11). The input embedding is denoted by E, the final hidden vector for the *[CLS]* token is denoted by $\boldsymbol{C} \in \mathbb{R}^H$, and the final hidden vector for the i^{th} input token is denoted by $\boldsymbol{T}_i \in \mathbb{R}^H$.

2.4.2 Pre-training BERT

BERT is pre-trained using two unsupervised tasks: Masked Language Model (MLM) and Next Sentence Prediction (NSP).

Masked Language Model

To train a deep bi-directional representation, a percentage of the input tokens are masked at random which are then predicted by the model. This procedure is often referred to as a *Cloze* task (Taylor [37]). The final hidden vectors of the mask tokens are passed through a *softmax* function over the vocabulary (corpus of tokens). A drawback of this approach is that there is a mismatch between pre-training and fine-tuning as the *[Mask]* token does not appear in fine-tuning. To remedy this, not all masked words are replaced by the *[Mask]* token. 15% of the token positions in the input sequence are chosen at random. If the i^{th} token is chosen, it is replaced with the *[Mask]* token 80% of the time, a random token 10% of the time, and the unchanged token 10% of the time. $\boldsymbol{T}_i$ is used to predict the original token using the cross-entropy loss function. Consider S length token sequences. Let $\hat{\mathbb{P}}\left[\boldsymbol{T}_{1:S}\right]$ denote a learned distribution over token sequences of length S. A probabilistic model $\hat{\mathbb{P}}\left[\boldsymbol{T}_i|\boldsymbol{T}_{<i},\boldsymbol{T}_{>i}\right]$ is implicitly defined by conditioning on the tokens before and after the i^{th} token. The cross-entropy objective function (Braverman et al. [4]) is defined by

$$CE\left[\mathbb{P}||\hat{\mathbb{P}}\right] \coloneqq \frac{1}{S}\underset{\boldsymbol{T}_{1:S}\sim\mathbb{P}}{\mathbb{E}}\left[\sum_{i=1}^{S}\log\left(\frac{1}{\hat{\mathbb{P}}\left[\boldsymbol{T}_i|\boldsymbol{T}_{<i},\boldsymbol{T}_{>i}\right]}\right)\right] = \frac{1}{S}\underset{\boldsymbol{T}_{1:S}\sim\mathbb{P}}{\mathbb{E}}\left[\log\left(\frac{1}{\hat{\mathbb{P}}\left[\boldsymbol{T}_{1:S}\right]}\right)\right]$$

where $\mathbb{P}$ is the true distribution of the tokens and $\mathbb{E}_{\boldsymbol{T}_{1:S}\sim\mathbb{P}}$ denotes the expectation under the true probability measure. Empirically, if $\log\left(\hat{\mathbb{P}}_{\boldsymbol{\theta}}\left[\boldsymbol{T}_i|\boldsymbol{T}_{<i},\boldsymbol{T}_{>i}\right]\right)$ is the log-likelihood of the i^{th} token conditioned on the preceding and following tokens according to our model,

$$CE\left(\left(\boldsymbol{T}_1,\cdots,\boldsymbol{T}_S\right)\right) \coloneqq -\frac{1}{S}\sum_{i=1}^{S}\log\left(\hat{\mathbb{P}}_{\boldsymbol{\theta}}\left[\boldsymbol{T}_i|\boldsymbol{T}_{<i},T_{>i}\right]\right).$$

Given M token sequences of length S, the cross-entropy loss function minimised by the model is

$$\mathcal{L}\left(\boldsymbol{\theta}\right) \coloneqq -\frac{1}{MS}\sum_{j=1}^{M}\sum_{i=1}^{S}\log\left(\hat{\mathbb{P}}_{\boldsymbol{\theta}}\left[\boldsymbol{T}_i^j|\boldsymbol{T}_{<i}^j,\boldsymbol{T}_{>i}^j\right]\right)$$

where $\boldsymbol{T}^j$ denotes tokens from the j^{th} sample.

The original token is kept 10% of the time to bias the representation towards the actual observed word. The main advantage of MLM is that the Transformer encoder will not know which words it needs to predict. This forces the Transformer encoder to keep a distributional contextual representation of every token. Since random replacement occurs 1.5% of the time, it does not damage the model's language understanding capabilities.

Next Sentence Prediction

Language modelling does not capture the relationship among sentences, which is essential to a significant number of NLP downstream tasks. To train the model to understand sentence relationships, the model is pre-trained using the next sentence prediction binary task. When choosing the two sentences A and B for the input representation, 50% of the time B is the sentence following sentence A (labelled as *IsNext*) whilst the other 50% of the time sentence B is a random sentence from the corpus (labelled as *NotNext*). The final hidden vector $\boldsymbol{C}$ is used as input into a binary classifier. The binary classifier predicts the label for sentence B.

2.4.3 Fine-tuning BERT

To fine-tune BERT, an additional model (such as linear layer or LSTM) is added on top of BERT which is specifically designed for the downstream task. This additional model is used to generate the desired output. The task-specific inputs and outputs are plugged into the model and the parameters are fine-tuned end-to-end. Sentences A and B from pre-training are analogous to question-answer pairs in question answering, sentence pairs in paraphrasing, etc. For classification tasks, A is the sentence to be classified whilst sentence B is the empty string $\emptyset$. The token representations are the input to the additional downstream specific model built on top of BERT.

Chapter 3

Implementation Details and Results

The objective of the model is to classify messages exchanged between organisations and stakeholders within the context of an organisation operating within the financial sector. The messages are exchanged via Bloomberg Terminals in relation to various industry related topics, such as government policies, trade ideas, and pricing models. By classifying the messages, the organisation will benefit from a better understanding of current topics that are generating interest amongst stakeholders.

3.1 Message Classification Model

The message classifier used for this study consists of two main parts; embeddings and classification. The embeddings were constructed using BERT. The BERT model used was pre-trained on a large text corpus using MLM and NSP as described in section 2.4.2. It consists of 12 stacked layers ($L = 12$), 12 attention heads per layer ($h = 12$), a hidden size of $H = 768$, and intermediate size $d_f = 3072$. The tokens were constructed using the WordPiece model along with a specific vocabulary text file containing a list of possible tokens. The embeddings constructed by BERT were then passed into a bi-LSTM model. The final embedding of the *[CLS]* token (after passing through the bi-LSTM) was passed as input to FNNs (a separate FNN for each label) which produced the final classification. Each FNN performed a binary classification task; whether the message falls into the current label or not. Each

FNN belongs to the following class

$$\mathcal{N}_2\left(H, H, 2; \mathbf{tanh}, \boldsymbol{Identity}\right)$$

where $\boldsymbol{Identity}: \mathbb{R}^2 \to \mathbb{R}^2$ is the identity function defined by $\boldsymbol{Identity}\left(x_1, x_2\right) = (x_1, x_2)$. The part of the model comprising of the bi-LSTM and FNNs is referred to as the classification head.

Training the model consisted of two steps. In the initial part of training, the classification head is pre-trained. Pre-training involved normal model training, however the parameters of the classification head are updated whilst the BERT parameters are not. The next part of training focused on fine-tuning the entire model (updating all the parameters). This was split into three stages where at each stage we start fine-tuning from the best model of the previous stage, i.e. fine-tuning stage i starts with model parameters initialised to the ones corresponding to the best model in stage $i-1$ for $i = 1, 2, 3$ where $i = 0$ corresponds to the pre-training of the classification head. The techniques used during pre-training and training are detailed in the following sections.

3.2 Optimisation and Learning Rate

Let $\hat{\boldsymbol{y}} = \left(\hat{\boldsymbol{y}}_{\mathbf{1}}, \cdots, \hat{\boldsymbol{y}}_{|\mathcal{Y}|}\right) \in \mathbb{R}^{|\mathcal{Y}| \times 2}$ be the output of the model where $\hat{\boldsymbol{y}}_i$ is the logit produced by the i^{th} classifier and $|\mathcal{Y}|$ denotes the amount of labels. Since each FNN in the classification head acts as a binary classifier with labels i^{th} *label* and *None*, $\hat{\boldsymbol{y}}_i \in \mathbb{R}^2$ is the unnormalised output of the FNN. The loss attributed to the i^{th} class is defined by

$$l\left(\hat{\boldsymbol{y}}_i, y_i\right) = -W_i \log\left(\frac{e^{\hat{\boldsymbol{y}}_{i,y_i}}}{e^{\hat{\boldsymbol{y}}_{i,1}} + e^{\hat{\boldsymbol{y}}_{i,2}}}\right)$$

where $\hat{\boldsymbol{y}}_{i,j}$ is the j^{th} element of $\hat{\boldsymbol{y}}_i$ for $j = 1, 2$, and $W_i \in \mathbb{R}$ is a weight, and $y_i \in \{0, 1\}$. In addition, an L1-regularization term was incorporated to regularize the non-bias parameters of the FNNs in the classification head. The total loss over a minibatch

B is the cross-entropy loss with regularization and is calculated using the formula

$$\mathcal{L}_B(\boldsymbol{\theta}) \coloneqq \lambda_{reg} \|\boldsymbol{U}\|_F - \frac{1}{|B|} \sum_{i \in B} \sum_{k=1}^{|\mathcal{Y}|} \boldsymbol{W}_k \log \left(\frac{e^{\hat{\boldsymbol{y}}_{i,k,\boldsymbol{y}_{i,k}}}}{e^{\hat{\boldsymbol{y}}_{i,k,1}} + e^{\hat{\boldsymbol{y}}_{i,k,2}}} \right) \tag{3.1}$$

where $\boldsymbol{y}_i \in \{0,1\}^{|\mathcal{Y}|}$ is the label vector of the i^{th} sample in the minibatch B, $\boldsymbol{W} \in \mathbb{R}^{|\mathcal{Y}|}$ is a weight vector, and an index of $\{i,j,k\}$ indicates the k^{th} value in the vector corresponding to the j^{th} class in the i^{th} sample. $\boldsymbol{U}$ is a matrix where each row is a flattened weight matrix parameter in the classification head, λ_{reg} is a regularization parameter, and $\|\cdot\|_F$ denotes the Frobenius norm of a matrix defined as

$$\|\boldsymbol{A}\|_F \coloneqq \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{i,j}|^2}, \quad \boldsymbol{A} = \{a_{i,j}\}_{i=1,\cdots,m, j=1,\cdots,n} \in \mathbb{R}^{m \times n}.$$

In our case $\boldsymbol{W} = \boldsymbol{1}_{|\mathcal{Y}|}$ and $\lambda_{reg} = 10^{-7}$.

The learning rate hyper-parameter is vital in controlling the learning of the system including generalisation and optimisation. In fact, Bengio et al. [2] describe it as the "*single most important hyper-parameter*". Therefore, it is vital that this parameter is chosen as optimally as possible.

When pre-training the classification head, the AdamW optimiser is used with $\epsilon = 10^{-4}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\lambda = 0.02$ for non-bias parameters, otherwise $\lambda = 0$. In this section, the AdamW hyper-parameter α is denoted by η_{-1}. 5 epochs are used along with minibatches of size 32. The learning rate is defined as a function of the current update step. The initial learning rate is $\eta_{-1} = \frac{250}{3} \cdot 10^{-5}$. The learning rate is then updated according to the cosine schedule given by Eq. (3.2). Let O be the total number of optimisation steps and $f_p \in [0,1]$ be a percentage of warm up steps. Let $W \coloneqq O \cdot f_p$ denote the number of warm up steps. The learning rate is updated according to the following schedule

$$\eta_t = \begin{cases} \eta_{-1} \frac{t}{max(1,W)}, & \text{if } t < W \\ \eta_{-1} max\left(0, \frac{1}{2}\left(1 + \cos\left(\frac{2\pi C(f_p,\eta)(t-W)}{max(1,O-W)}\right)\right)\right), & \text{otherwise} \end{cases} \tag{3.2}$$

where $f_{p,\eta} \in [0, 1]$ is a hyper-parameter controlling the number of cycles in the cosine schedule, and $C: \mathbb{R} \to \mathbb{R}$ is defined by

$$C(x) = \frac{1}{2\pi} \arccos(2x - 1).$$

Intuitively, the learning rate is increased linearly from 0 to $\eta_W = \eta_{-1}$ over the optimisation steps 0 to W and then decreased following a cosine function from η_W to η_O. In our case we set $f_p = 0.05$ and $f_{p,\eta} = 0.2$. Adding gradual warm up to the learning rate schedule is presented in Goyal et al. [12] and certain benefits of decaying the learning rate (optimisation steps after step W) are discussed in You et al. [42].

A similar approach to pre-training the classification head was adopted for the fine-tuning process. As previously mentioned, fine-tuning was performed in three stages of 20, 15, and 15 epochs respectively. The batch size was set to 8 and an AdamW optimiser with $\epsilon = 10^{-4}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ was used (for all stages) to minimise the loss function. A major difference to the pre-training of the classification head is that different parts of the model have their own specific learning rates. This is an optimisation technique described in Singh et al. [35]. Let f_d be a hyper-parameter controlling the learning rate decay and η'_{-1} be a common learning rate hyper-parameter. The learning rate for the i^{th} encoder layer was set to

$$\eta_{-1,encoder} = \eta'_{-1} f_d^{L-i}$$

for $i = 1, \cdots, 12$. The learning rate is shown in Figure 3.1.

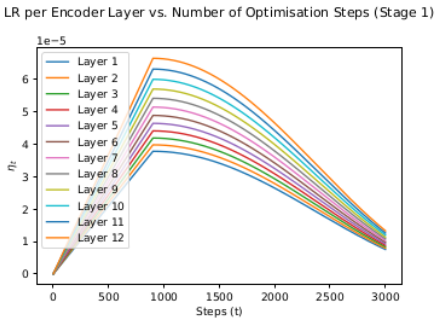


Figure 3.1: Learning rate of each encoder layer as a function of the number of optimisation steps.

The learning rate for the parameters in the embedding layers was set to

$$\eta_{-1,embedding} = \eta'_{-1} f_d^{L+1}$$

and the learning rate for all other parameters was set to η'_{-1}. The learning rates were updated according to the cosine schedule (Eq. (3.2)). We set $f_{p,\eta} = [0.2, 0.02, 0.2]$, $\eta'_{-1} = [7 \cdot 10^{-5}, 2.5 \cdot 10^{-5}, 2.5 \cdot 10^{-5}]$, $f_p = [0.3, 0.05, 0.01]$, $f_d = [0.95, 0.95, 0.95]$ where the i^{th} item in the list corresponds to the i^{th} stage. Bias vectors and layer normalisation hyper-parameters had $\lambda = 0$, whilst all other parameters had $\lambda = 0.02$.

3.3 Dropout and Zoneout

Deep neural networks are capable of learning many complex relationships between their inputs and outputs. However, with limited training data available, many of these relationships form as a result of sampling noise present in the training data. Therefore, they will not be present in the testing data even if the testing data is sampled from the same distribution as the training data. This results in overfitting, a recurring problem in machine learning tasks. Many techniques have been developed to reduce overfitting. One way of handling overfitting is to stop training when performance on the validation set starts to decrease. A popular technique which is known to improve machine learning tasks is combining various models. In the case of neural networks, this process is very expensive and not always feasible. Dropout is a technique developed by Srivastava et al. [36] which prevents overfitting of neural networks. It also provides a way of approximately merging exponentially many different neural network architectures. Dropout consists of *dropping* units (hidden and input) in a neural network. By *dropping* a unit, we mean temporarily removing the unit from the network along with any incoming and outgoing connections, as shown in Figure 3.2. The choice of which units to drop is performed at random by a dropout layer. A common approach is retaining a unit with probability $p_{dropout}$. The dropout layer corresponding to the l^{th} layer of a FNN with an element-wise activation function is described by the following equations

$$\boldsymbol{r}^l \coloneqq \left(r_1^l, \cdots, r_{d_l}^l\right), \quad r_j^l \sim Bernoulli\left(p\right), j = 1, \cdots, d_l$$

$$\tilde{\boldsymbol{y}}^l \coloneqq \boldsymbol{r}^l \odot \boldsymbol{y}^l$$
$$z_i^{l+1} = \boldsymbol{W}_i^{l+1}\tilde{\boldsymbol{y}}^l + b_i^{l+1}$$
$$y_i^{l+1} = \sigma\left(z_i^{l+1}\right)$$

where $\boldsymbol{r}^l$ is a vector of independent and identically distributed Bernoulli random variables, $\boldsymbol{z}^{l+1}$ is the input to the $(l+1)^{\text{th}}$-layer's activation function, $\boldsymbol{W}_i^{l+1}$ is the i^{th} row of the weight matrix, b_i^{l+1} is the i^{th}-entry of the bias vector, $\boldsymbol{y}^l$ is the output of the l^{th} layer and σ is an activation function.

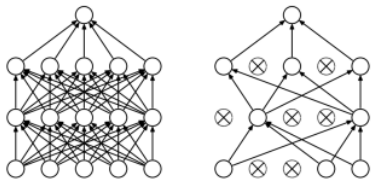


Figure 3.2: Graphical Representation of dropout (Srivastava et al. [36, Section 1 *Introduction*, Figure 1, page 1930]). **Left:** Standard neural network with 2 hidden layers. **Right:** Resulting neural network after applying dropout to the network on the left. Crossed units have been dropped.

A technique used to prevent overfitting in RNNs and which is similar to dropout is zoneout (Krueger et al. [23]). In zoneout, noise is injected into the model during training, however, instead of dropping out layers, units are *zoned out* and set to their previous value ($\boldsymbol{h}^t = \boldsymbol{h}^{t-1}$). The objective of zoneout is to improve the RNN's robustness to perturbations in the hidden state with the effect of regularizing the dynamics. Zoneout also helps in combating the vanishing gradient problem and is more appealing than dropout since it preserves information flow forwards and backwards.

In this research, dropout layers were placed throughout the model. A dropout layer was placed before the encoder. In each encoder layer, a dropout layer was placed after the *softmax* function is applied in the scaled dot-product attention heads, after concatenating all the attention heads, and at the output of the encoder layer. During pre-training of the classification heads, the dropout probabilities were set to 0.1. In the fine-tuning phase of training, the dropout probabilities were set to 0.15. Another dropout layer with $p = 0.6$ (for pre-training and fine-tuning) was placed

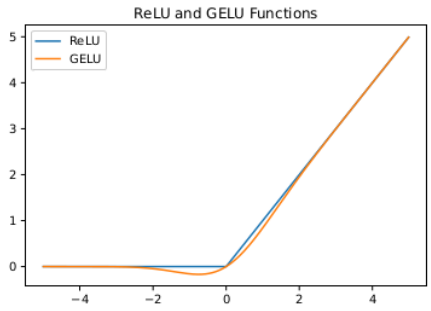


Figure 3.3: ReLU and GELU functions.

between the two layers of each FNN in the classification head.

3.4 Gaussian Error Linear Unit

The purpose of the activation function is to add non-linearity into the network. This non-linearity is kept separate from the stochastic regularizers such as dropout, even though they both determine the output of the layer. The Gaussian Error Linear Unity (GELU) (Hendrycks and Gimpel [16]) combines a non-linearity activation function with stochastic regularization. GELU is formulated by combining the ReLU activation function with properties of dropout and zoneout. The GELU function multiplies the input by zero or one, where the values of the zero-one mask are stochastically determined whilst being jointly dependent on the input. The input x is multiplied by $m \sim Bernoulli\left(\Phi\left(x\right)\right)$, where $\Phi\left(x\right) = \mathbb{P}\left[X < x\right]$, $X \sim \mathcal{N}\left(0, 1\right)$ is the cumulative distribution function (CDF) of the standard normal distribution. Under this function, inputs have a higher probability of being dropped as x decreases (Figure 3.3). Therefore, the GELU function is stochastic as well as dependent on the input. Since the CDF of a Gaussian random variable is often computed with the error function $\text{erf}\left(z\right) \coloneqq \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$, GELU is defined as follows

$$GELU\left(x\right) \coloneqq x\mathbb{P}\left[X < x\right] = x\Phi\left(x\right) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right].$$

To benefit from the advantages outlined above, each ReLU activation function in the model is replaced by the GELU activation function.

3.5 Dataset and Results

The dataset consisted of 1550 text passages. The text data was accumulated from two sources; Bloomberg Terminal chat messages and research reports. The text passages were classified into 13 labels (Table 3.1). These labels were chosen as they provide useful information on the current market and client landscape. An inherent characteristic of the dataset is that the number of positive labels per class (Table 3.1) is low relative to the number of samples. Another feature of the dataset is that a text passage can be associated with multiple labels. For example, the message "Bank of England maintains interest rate" is classified as a *Central Bank*, *Policy Rates*, and *Monetary Policy* message. Another characteristic of the dataset is that certain labels tend to aggregate together, for example *Monetary Policy* and *Policy Rates* are more likely to be associated with the same text passage than *Policy Rates* and *Trade Recommendation*. Labelling the dataset was performed in an iterative process consisting of two steps. The first step involved manually tagging the text passages. The next step involved building a simple model to tag new data and check the newly tagged data. The model built used BOW embeddings as inputs to 13 support vector machines (SVM) (described in Appendix D), one for each label. Each SVM acted as a binary classifier for a particular label. Once the model was run on new data, the tagging was checked and the new data samples were added to the existing dataset. This process was repeated until a sufficient number of samples was tagged. Using a model to tag the data provided insight into whether a functional relationship between the text passages and labels could be found. Although the model is relatively simple, this provided valuable insight into the dataset through the keywords identified by the model for each category. For example, the model rated "ecb" for *Central Bank* ("ecb" is an acronym for European Central Bank) and "qe" for *Monetary Policy* ("qe" is an acronym for quantitative easing) as *strong* words in these respective categories. Other categories such as *Request For Quote* (*RFQ*) messages, were always tagged as not falling into any category since a requirement of the model was to also identify messages that were not of interest to us.

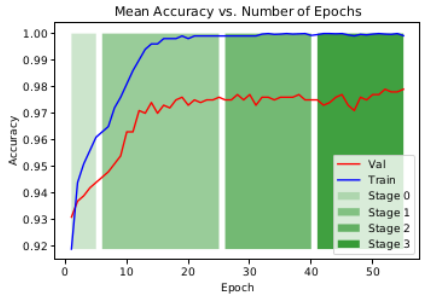
To test the model, the dataset was randomly split into a training set and a val-

idation set. Due to the class imbalance of each label (many more negatives than positives per label), the random split was subject to the constraint that each label had more positives in the training set and there were positives in the validation set. The training dataset had 1200 training instances and 350 validation instances. The number of positive instances per label are provided in Table 3.1.

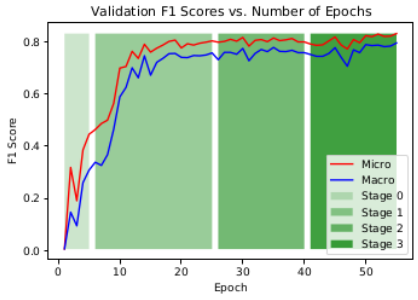
Label	**# Train Pos.**	**# Valid. Pos.**	**# Pred. TPs**
Central Bank	142	58	54
Interest & FX Rates	123	35	18
Policy Rates	51	20	15
Inflation	40	15	14
Monetary Policy	114	50	42
Fiscal Policy	22	11	7
Central Bank Meeting	46	10	6
Economic Forecast	50	12	8
Bonds	60	23	18
Trade Recommendation	57	21	8
Emerging Markets	88	20	11
Curve	72	29	27
Bond Yields	20	7	6

Table 3.1: Number of Positive Instances Per Class in the Training and Validation Sets.

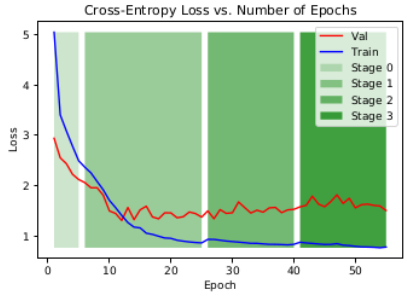
To evaluate model performance, four different values were computed on the validation set. The mean validation accuracy was computed by averaging the accuracies on the validation set of the binary classifiers. Although this gave a good indication of model improvement and convergence, it can be misleading due to the imbalances in each label. Suppose 1 indicates a positive prediction and 0 a negative one. If a label has 95% of the values in the validation set as 0 and the rest as 1, predicting all 0 will result in a validation accuracy of 95%. Two metrics which are more indicative of model performance are the Micro and Macro F1 Scores (Zhang and Zhou [43]). Let TP_i, FP_i, FN_i denote the number of true positives, false positives, and false negatives of the i^{th} binary classifier respectively for $i = 1, \cdots, |\mathcal{Y}|$. The Micro and



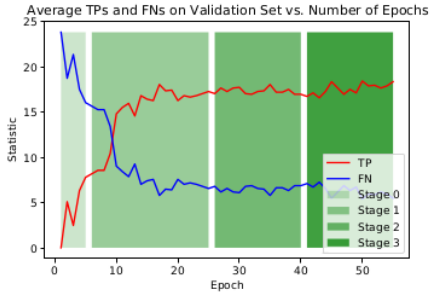
(a) Mean accuracy of the validation and training sets.



(b) Validation F1 Scores.



(c) Training and Validation cross-entropy loss.



(d) Average Validation TPs and FNs.

Figure 3.4: Evaluation Metrics. The green shading reflects the training stage.

Macro F1 Scores are computed using the formulae

$$F^1_{micro} := \frac{2\sum_{i=1}^{|\mathcal{Y}|} TP_i}{2\sum_{i=1}^{|\mathcal{Y}|} TP_i + \sum_{i=1}^{|\mathcal{Y}|} FN_i + \sum_{i=1}^{|\mathcal{Y}|} FP_i}, \quad F^1_{macro} := \frac{1}{|\mathcal{Y}|}\sum_{i=1}^{|\mathcal{Y}|} \frac{2TP_i}{2TP_i + FN_i + FP_i}.$$

Since the Micro and Macro F1 scores are computed using the true positives, false negatives, and false positives per label, they do not suffer from the problem described for the mean validation accuracy. The final metric used to evaluate the model is validation loss calculated using Eq. (3.1) with $\lambda_{reg} = 0$ and a batch size of 64.

The results (Figure 3.4) show that the model did improve and converge. The mean validation accuracy (Figure 3.4a) starts off high at 0.931 and reaches a maximum of 0.979. The main reason for such a high initial value is that most of the predictions were negative, resulting in the problem described above for the mean validation accuracy metric. This is also reflected in the graph showing the average number

of TPs and FNs (Figure 3.4d). Initially, the average amount of FNs is relatively high and decreases whilst the average amount of TPs increases. The averages are quite low relative to the number of validation instances because of the positive label imbalance across labels. Although the graph of mean validation accuracy (Figure 3.4a) illustrates the *learning* of the model, this is better captured in the graph of the F1 scores (Figure 3.4b) since there is a larger improvement in the scores. Both F1 Scores are contained within the range $[0, 1]$ and the higher the score the better the model. As they are computed using the TPs, FPs, and FNs the scores give a clear picture of model performance. To calculate the Micro F1 Score, we aggregate the contributions of all labels to compute the final score. On the other hand, to compute the Macro F1 Score, the ratio $\frac{2TP}{2TP+FN+FP}$ is computed independently for each label and then averaged. Therefore, when computing the Micro F1 Score all labels are treated equally whilst in the Macro F1 Score the labels with the least positives are given a higher weighting. In terms of scores produced, the Micro F1 score assumes no label is more important than another and therefore the score focuses on the number of correct predictions. On the other hand, the Macro F1 Score is insensitive to class imbalances and therefore gives a higher weighting to the class with the least positives. In multi-label problems with an imbalance of positive labels across classes, the Micro F1 Score is generally the preferred metric. On the validation set, both the Micro and Macro F1 scores improved drastically. They both converged to values of around 0.8 with the Micro F1 score constantly above the Macro F1 score as from the second epoch. The same information can be extrapolated from the cross-entropy loss graph (Figure 3.4c). The model picked up on the acronyms *boe* and *qe* for central bank and monetary policy respectively as they featured frequently in the training data. Analysing the results produced, the labels with the worst performance are the ones with the most varied language. Not all emerging market countries were in the training set and trade recommendation jargon changed across messages making them more difficult to tag. This resulted in these categories performing slightly worse compared to other categories. Another limitation of the model was picking up on currency pairs and unseen currencies. This could be resolved in two ways; either adding more data to the training set or fine-tuning the embeddings using the current dataset to the MLM task. To fine-tune the embeddings using the MLM

task we would mask tokens based on a probability distribution with the tokens concerning currencies having a higher probability of being masked. This would have improved the encodings of specific finance-related keywords. The new fine-tuned parameters would be used as the initial parameters in the message classification model.

The simple model used to tag the data was also run on the final dataset. The simple model performed well in the *Central Bank* category correctly predicting 46 positives out of 58 whilst the BERT model correctly predicted 54 positives. The SVM classifier identified keywords such as "ecb" and "central". Due to the limitation in the BOW encodings and the simplicity of SVM classifiers, the model performed best in categories with easily identifiable and recurring keywords (such as *Central Bank*). For example, for the *Bonds* label, the simple model correctly predicted 8 out of 23 positives, half the amount correctly predicted by the BERT model. This outlines one of the major advantages of BERT over the BOW encodings. *Bond* was one of the keywords given a high weighting by the simple classifier. If this keyword were used in a message such as "can I see your prices on bond", since context is not considered by the simple model this message is tagged as a *Bonds* message although it is an *RFQ*. This problem is alleviated by using the BERT encodings. This limitation of the simple model was reflected in other labels. Out of 10 positives in the *Central Bank Meeting* label, the simple model predicted 3 correct positives and the BERT model correctly predicted 7 positives. The most prominent keywords identified by the simple model were *fomc* (acronym for Federal Open Market Committee) and *meet*. These should be helpful when classifying messages such as "the minutes and summary of the bank of england monetary policy committee's march meeting" as it contains the word "meeting". On the other hand, the simple model has trouble generalising and understanding the content of the message correctly when the keywords are not present. However, the BERT model correctly predicted *Central Bank Meeting* messages such as "the central bank of Egypt cbe kept its policy rates on hold at its mpc this afternoon". The BERT model predicts this message correctly by associating the phrase "central bank of Egypt" with the *Central Bank* part of the *Central Bank Meeting* label and identifies the *Meeting* aspect of the message from "mpc this afternoon" ("mpc" is an acronym for Monetary Policy Committee).

To try and improve model performance, context was added to the messages. For each tagged message, the previous and following messages were added to the instance and the tagged message was enclosed in a special token, i.e. if B is the tagged message, A is the previous message, and C is the following message, the new data instance is A *[ST]* B *[ST]* C. The messages A or C can be empty strings (for example, if B is the last message in the conversation, C will be empty). The model with context performed slightly worse than the model without context. The highest mean validation accuracy and Micro F1 Score the model with context managed to achieve were 0.971 and 0.764 respectively. A reason for this dip in performance could be that since the topic of conversation changes quickly when exchanging messages, the previous and following messages could not be related to the message of interest (message B). In this case, messages A or C are filling the dataset with noise. For example, suppose message B is "Bostic says he upgraded his 2021 inflation forecast" whilst message A is "Good morning" and message C is an empty string. The new instance is

"Good morning *[ST]* Bostic says he upgraded his 2021 inflation forecast *[ST]* $\emptyset$".

However, message A provides no context to assist in tagging message B. Moreover, suppose *A1 [ST] B [ST] C1* and *A2 [ST] A1 [ST] C2* are two instances in the dataset. If the labels associated with the first message are different to the second, the model is fed message *A1* with two different labels. This could cause contradictions for the model. For example, consider the following two instances

"focus remained on the gbp *[ST]* policymakers caution in dialling back fiscal stimulus *[ST]* $\emptyset$"

and

"$\emptyset$ *[ST]* focus remained on the gbp *[ST]* policymakers caution in dialling back fiscal stimulus".

The first instance is tagged as *Fiscal Policy* whilst the second instance is tagged as *Interest & FX Rates*. Since "focus remained on gbp" and "polciymakers caution in dialling back fiscal stimulus" are in both instances and they have different labels, the classifier might get confused as to the correct tags. This contradiction occurred in 267 instances.

Conclusion

In this thesis we focused on classifying financial messages. Classifying messages posed multiple challenges such as encoding the text data, tagging the data, and handling the class imbalances. The model used to embed the data was BERT. A classification head consisting of a bi-LSTM and a FNN for each label was added on top of BERT. The model was trained by minimising cross-entropy loss and the weights were updated using AdamW.

Multiple techniques were used to prevent overfitting and improve model convergence and performance. For example, the classification head was pre-trained, fine-tuning consisted of multiple stages, the learning rate followed a cosine schedule, and different layers had different learning rates. Results showed that the model performed very well, with the model achieving a mean validation accuracy of 0.979 and a micro F1 Score of above 0.8. This showed that the class imbalance was handled well by the model. Convergence is visible in the figures presented. When context was added to the instances, the model performed slightly worse.

Further research could focus on improving the BERT encodings. This could be done by fine-tuning BERT to the MLM task using the current dataset as described in Section 3.5. Another area of further research could concentrate on incorporating the Sig-Transformer Encoder into the classifier as described in Biyong et al. [32]. In addition, identifying important Attention heads and pruning unnecessary ones (Voita et al. [39]) is an area which could also be studied further in the context of this message classifier.

Appendix A

Technical Proofs and Derivations

A.1 FNN Backpropagation

Proof of the backpropagaion of FNNs (Theorem 8).

Proof. ([29, Section 3 *Training feedforward neural networks*, pages 39-40]) Since $l = l\left(\boldsymbol{a}^r, \boldsymbol{y}\right) = l\left(\boldsymbol{\sigma}_r\left(\boldsymbol{z}^r\right), \boldsymbol{y}\right)$ we apply to chain rule to obtain

$$\delta_j^r = \frac{\partial l}{\partial z_j^r} = \sum_{k=0}^{N} \frac{\partial l}{\partial \hat{y}_k}\left(\boldsymbol{a}^r, \boldsymbol{y}\right) \frac{\partial a_k^r}{\partial z_j^r} = g_r'\left(z_j^r\right) \frac{\partial l}{\partial \hat{y}_j}\left(\boldsymbol{a}^r, \boldsymbol{y}\right), \quad j = 1, \cdots, N \tag{A.1}$$

since

$$\frac{\partial a_k^r}{\partial z_j^r} = \begin{cases} g_r'\left(z_j^r\right), & k = j \\ 0, & k \neq j \end{cases}.$$

Equation (2.6) is equation (A.1) in vector form.

To obtain (2.7), we first apply the chain rule to δ_j^i

$$\delta_j^i = \frac{\partial l}{\partial z_j^i} = \sum_{k=1}^{d_{i+1}} \frac{\partial l}{\partial z_k^{i+1}} \frac{\partial z_k^{i+1}}{\partial z_j^i} = \sum_{k=1}^{d_{i+1}} \delta_k^{i+1} \frac{\partial z_k^{i+1}}{\partial z_j^i}, \quad j = 1, \cdots, d_i. \tag{A.2}$$

Since,

$$z_k^{i+1} = \sum_{u=1}^{d_i} W_{k,u}^{i+1} g_i\left(z_u^i\right) + b_k^{i+1} \tag{A.3}$$

it follows that

$$\frac{\partial z_k^{i+1}}{\partial z_j^i} = W_{k,j}^{i+1} g_i' \left(z_j^i\right).$$

Then

$$\delta_i^j = g_i' \left(z_j^i\right) \sum_{k=1}^{d_{i+1}} \delta_k^{i+1} W_{k,j}^{i+1} = g_i' \left(z_j^i\right) \sum_{k=1}^{d_i} \left(W^{i+1}\right)_{j,k}^T \delta_k^{i+1}, \quad j = 1, \cdots, d_i$$

which is (2.7) in vector/matrix form.

Replacing $i+1$ with i in (A.3), we get

$$\frac{\partial z_k^i}{\partial b_j^i} = \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases} \text{ and } \frac{\partial z_k^i}{\partial W_{j,u}^i} = \begin{cases} a_u^{i-1}, & k = j \\ 0, & k \neq j \end{cases}.$$

Applying the chain rule we get

$$\frac{\partial l}{\partial b_j^i} = \sum_{k=1}^{d_i} \frac{\partial l}{\partial z_k^i} \frac{\partial z_k^i}{\partial b_j^i} = \delta_j^i \text{ and } \frac{\partial l}{\partial W_{j,u}^i} = \sum_{k=1}^{d_i} \frac{\partial l}{\partial z_k^i} \frac{\partial z_k^i}{\partial W_{j,u}^i} = \delta_j^i a_u^{i-1}.$$

□

A.2 SDEs of Optimisation Algorithms

To analyse the SDEs, we first need the following result.

Theorem 11. *Let $\boldsymbol{X} \sim S\alpha S\left(a\boldsymbol{I}\right)$ for $a > 0$. Then $\boldsymbol{X} = aS\alpha S\left(\boldsymbol{I}\right)$ in distribution.*

Proof. Let $\boldsymbol{X} = (X_1, \cdots, X_N) \sim \alpha S\alpha S(a\boldsymbol{I})$. Since the covariance matrix is $a\boldsymbol{I}$, X_i are independent and $X_i \sim S\alpha S\left(a\right)$ for $i = 1, \cdots, N$. Using the independence properties of the characteristic function,

$$\phi_{\boldsymbol{X}}\left(\boldsymbol{w}\right) = \prod_{j=1}^{N} \phi_{X_j}\left(w_j\right) = \prod_{j=1}^{N} \exp\left(-\left|aw_j\right|^{\alpha}\right).$$

Similarly, let $\boldsymbol{Y} = (Y_1, \cdots, Y_N) \sim aS\alpha S\left(\boldsymbol{I}\right)$. The Y_i are independent and $Y_i \sim aS\alpha S\left(1\right)$ for $i = 1, \cdots, N$. Applying the scaling and independence properties of the

characteristic function we get

$$\phi_{\boldsymbol{Y}}(\boldsymbol{w}) = \prod_{j=1}^{N} \phi_{Y_j}(w_j) = \prod_{j=1}^{N} \phi_Z(aw_j) = \prod_{j=1}^{N} \exp\left(-|aw_j|^{\alpha}\right) = \phi_{\boldsymbol{X}}(\boldsymbol{w}),$$

where $Z \sim S\alpha S(1)$. Since the characteristic functions are equal, $\boldsymbol{X} = \boldsymbol{Y} = aS\alpha S(\boldsymbol{I})$ in distribution. □

A.2.1 Lévy-driven SDE for SGD

The Lévy-driven SDE for SGD is given by

$$d\boldsymbol{\theta}_i = -\nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*(\theta_i)\, di + \epsilon \boldsymbol{\Sigma}_i d\boldsymbol{L}_i.$$

Consider the grid points $t_0 < t_1 < \cdots < t_N$ for some $N \in \mathbb{N}$ such that $t_i = i\eta$ for $i = 0, \cdots N$. Now

$$\int_{i\eta}^{(i+1)\eta} -\nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*(\boldsymbol{\theta}_i)\, di \approx -\eta \nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*(\boldsymbol{\theta}_{i\eta})$$

and

$$\int_{i\eta}^{(i+1)\eta} \epsilon \boldsymbol{\Sigma}_i d\boldsymbol{L}_i \approx \epsilon \boldsymbol{\Sigma}_{i\eta} \left(\boldsymbol{L}_{(i+1)\eta} - \boldsymbol{L}_{i\eta}\right).$$

By definition of Lévy motion,

$$\left(\boldsymbol{L}_{(i+1)\eta} - \boldsymbol{L}_{i\eta}\right) \overset{D}{=} S\alpha S\left(\left((i+1)\eta - i\eta\right)^{1/\alpha} \boldsymbol{I}\right) = S\alpha S\left(\eta^{1/\alpha} \boldsymbol{I}\right).$$

Discretising the SDE by applying the Euler scheme and applying Theorem 11, we get

$$\boldsymbol{\theta}_{(i+1)\eta} - \boldsymbol{\theta}_{i\eta} = -\eta \nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*(\boldsymbol{\theta}_{i\eta}) + \eta^{1/\alpha} \epsilon \boldsymbol{\Sigma}_{i\eta} \boldsymbol{L}_{i\eta}.$$

Since $\epsilon = \eta^{(\alpha-1)/\alpha}$ and the gradient noise is $S\alpha S(\boldsymbol{\Sigma}_i)$-distributed, by relabelling $i\eta$ with i and $(i+1)\eta$ with $i+1$ we get the SGD update given by Eq. (2.4).

A.2.2 Lévy-driven SDE for Adam

By relabelling the indices, the Adam update scheme can be formulated as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \hat{\boldsymbol{m}}_i / \left(\sqrt{\hat{\boldsymbol{v}}_i} + \epsilon\right)$$

where $\eta > 0$ is the learning rate,

$$\hat{\boldsymbol{m}}_i = \frac{1}{1-\beta_1^i}\boldsymbol{m}_i, \quad \boldsymbol{m}_i = \beta_1 \boldsymbol{m}_{i-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}}\mathcal{L}_{B_i}(\boldsymbol{\theta}_i),$$

and

$$\hat{\boldsymbol{v}}_i = \frac{1}{1-\beta_2^i}, \quad \boldsymbol{v}_i = \beta_2 \boldsymbol{v}_{i-1} + (1-\beta_2)\left(\mathcal{L}_{B_i}(\boldsymbol{\theta}_i) \odot \mathcal{L}_{B_i}(\boldsymbol{\theta}_i)\right).$$

By definitions of $\boldsymbol{m}_i$ and $\boldsymbol{m}_i'$, we get

$$\begin{aligned}
\boldsymbol{m}_i' - \boldsymbol{m}_i &= \beta_1 \boldsymbol{m}_{i-1}' + (1-\beta_1)\nabla_{\boldsymbol{\theta}} \boldsymbol{f}^*(\boldsymbol{\theta}_i) - \beta_1 \boldsymbol{m}_{i-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}}\mathcal{L}_{B_i}(\boldsymbol{\theta}_i) \\
&= (1-\beta_1)\sum_{j=0}^{i} \beta_1^{i-j}\left[\nabla_{\boldsymbol{\theta}}\boldsymbol{f}^*(\boldsymbol{\theta}_j) - \nabla_{\boldsymbol{\theta}}\mathcal{L}_{B_j}(\boldsymbol{\theta}_j)\right] \\
&= (1-\beta_1)\sum_{j=0}^{i} \beta_1^{i-j}\boldsymbol{u}_j.
\end{aligned}$$

Assume $\frac{1}{1-\beta_1^i}(\boldsymbol{m}_i' - \boldsymbol{m}_i) \sim S\alpha S(\boldsymbol{I})$ with covariance matrix $\boldsymbol{\Sigma}_i$. Since

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \eta \boldsymbol{m}_{i-1}'/\boldsymbol{z}_{i-1} + \eta\left(\boldsymbol{m}_{i-1}' - \boldsymbol{m}_{i-1}\right)/\boldsymbol{z}_{i-1}$$

where

$$\boldsymbol{z}_{i-1} \coloneqq \left(1-\beta_1^{i-1}\right)\left(\sqrt{\hat{\boldsymbol{v}}_{i-1}} + \epsilon\right),$$

it follows that the Lévy-driven SDE for Adam is given by

$$\begin{aligned}
d\boldsymbol{\theta}_i = -\mu_i \boldsymbol{Q}_i^{-1}\boldsymbol{m}_i' di + \epsilon \boldsymbol{Q}_i^{-1}\boldsymbol{\Sigma}_i d\boldsymbol{L}_i, \quad d\boldsymbol{m}_i' &= \beta_1\left(\nabla_{\boldsymbol{\theta}}\boldsymbol{f}^*(\boldsymbol{\theta}_i) - \boldsymbol{m}_i'\right)di, \\
d\boldsymbol{v}_i &= \beta_2\left[\nabla_{\boldsymbol{\theta}}\mathcal{L}_{B_i}(\boldsymbol{\theta}_i) \odot \nabla_{\boldsymbol{\theta}}\mathcal{L}_{B_i}(\boldsymbol{\theta}_i) - \boldsymbol{v}_i\right]di.
\end{aligned}$$

where $\epsilon = \eta^{(\alpha-1)/\alpha}$, $\boldsymbol{Q}_i = diag\left(\sqrt{w_i \boldsymbol{v}_i} + \epsilon\right)$, $u_i = 1/\left(1-e^{-\beta_1 i}\right)$, and $w_i = 1/\left(1-e^{-\beta_2 i}\right)$ are two constants which correct the bias in $\boldsymbol{m}_i'$ and $\boldsymbol{v}_i$.

A.3 Layer Normalisation

Since no proof of Theorem 9 was provided in Ba et al. [1], one is provided below.

Proof. Let a_j^{l-1} and γ_j denote the j^{th} entry in $\boldsymbol{a}^{l-1}$ and $\boldsymbol{\gamma}$ respectively. Also, let $w_{k_{i,j}}^l$

denote the $(i,j)^{\text{th}}$ entry of the matrix $\boldsymbol{W}_k^l$ for $k = 1, 2$. We have that,

$$\begin{aligned}
\mu_2^l = d_l^{-1} \sum_{i=1}^{d_l} q_{2_i}^l &= d_l^{-1} \sum_{i=1}^{d_l} \sum_{j=1}^{d_{l-1}} w_{2_{i,j}}^l a_j^{l-1} \\
&= d_l^{-1} \sum_{i=1}^{d_l} \sum_{j=1}^{d_{l-1}} \left(\delta w_{1_{i,j}}^l + \gamma_j \right) a_j^{l-1} \\
&= d_l^{-1} \left(\sum_{i=1}^{d_l} \sum_{j=1}^{d_{l-1}} \delta w_{1_{i,j}}^l a_j^{l-1} + \sum_{i=1}^{d_l} \sum_{j=1}^{d_{l-1}} \gamma_j a_j^{l-1} \right) \\
&= \delta \mu_1^l + \boldsymbol{\gamma}^T \boldsymbol{a}^{l-1}.
\end{aligned}$$

Similarly

$$\begin{aligned}
\left(\sigma_2^l\right)^2 = d_l^{-1} \sum_{i=1}^{d_l} \left(q_{2_i}^l - \mu_2^l\right)^2 &= d_l^{-1} \sum_{i=1}^{d_l} \left[\left(\sum_{j=1}^{d_{l-1}} \left(\delta w_{1_{i,j}}^l + \gamma_j \right) a_j^{l-1} \right) - \mu_2^l \right]^2 \\
&= d_l^{-1} \sum_{i=1}^{d_l} \left[\left(\sum_{j=1}^{d_{l-1}} \left(\delta w_{1_{i,j}}^l + \gamma_j \right) a_j^{l-1} \right) - \delta \mu_1^l + \boldsymbol{\gamma}^T \boldsymbol{a}^{l-1} \right]^2 \\
&= d_l^{-1} \sum_{i=1}^{d_l} \left[\left(\sum_{j=1}^{d_{l-1}} \delta w_{1_{i,j}}^l a_j^{l-1} \right) + \boldsymbol{\gamma}^T \boldsymbol{a}^{l-1} - \delta \mu_1^l - \boldsymbol{\gamma}^T \boldsymbol{a}^{l-1} \right]^2 \\
&= d_l^{-1} \sum_{i=1}^{d_l} \delta^2 \left(q_{1_i}^l - \mu_1^l\right)^2 \\
&= \left(\delta \sigma_1^l\right)^2.
\end{aligned}$$

It follows that

$$\begin{aligned}
\boldsymbol{a}_2^l &= \boldsymbol{\sigma}^l \left(\frac{\boldsymbol{g}^l}{\sigma_2^l} \left(W_2^l \boldsymbol{a}^{l-1} - \mu_2^l \mathbf{1}_{d_l} \right) + \boldsymbol{b}^l \right) \\
&= \boldsymbol{\sigma}^l \left(\frac{\boldsymbol{g}^l}{\sigma_2^l} \left(\left(\delta \boldsymbol{W}_1^l + \mathbf{1}_{d_l} \boldsymbol{\gamma}^T \right) \boldsymbol{a}^{l-1} - \mu_2^l \mathbf{1}_{d_l} \right) + \boldsymbol{b}^l \right) \\
&= \boldsymbol{\sigma}^l \left(\frac{\boldsymbol{g}^l}{\delta \sigma_1^l} \left(\delta \boldsymbol{W}_1^l \boldsymbol{a}^{l-1} - \delta \mu_1^l \mathbf{1}_{d_l} \right) + \boldsymbol{b}^l \right) \\
&= \boldsymbol{a}_1^l.
\end{aligned}$$

□

Appendix B

WordPiece Tokenization Algorithm

Algorithm 2: WordPiece Tokenizer. A message is represented as a list of tokens and each token is represented as a list of characters. Empty lists are denoted by $[\emptyset]$.

Require: Vocabulary File: *vocab.txt*
Input: Message $[M]_{i=1,\cdots,S}$
$outputTokens = [\emptyset]$
for *Token* T *in* $[M]_{i=1,\cdots,S}$ **do**
  $N = Length\,(T)$
  $characters = [T_j]_{j=1,\cdots,N}$
  if $N > 1000$ **then**
    Append "[UNK]" to *outputTokens*
  $start = 0$
  $isBad = False$
  $subTokens = [\emptyset]$
  while $start < N$ **do**
    $end = N$
    $currentSubstring = [\emptyset]$
    while $start < end$ **do**
      $substring = T_{start}, \cdots, T_{end-1}$
      if $start > 0$ **then**
        $substring =$ "##" $+substring$
      if $substring \in vocab.txt$ **then**
        $currentSubstring = substring$
        break
      $end = end - 1$
    if $currentSubstring = [\emptyset]$ **then**
      $isBad = True$
      break
    Append $currentSubstring$ to $subTokens$
    $start = end$
  if $isBad = True$ **then**
    Append "[UNK]" to *outputTokens*
  else
    Place $subTokens$ at the end of $outputTokens$
Output: $outputTokens$

Appendix C

Computing Multi-Head Attention

The first step in computing the multi-head attention involves constructing the query, key, and value vectors. To speed up computations, the query, key, and value vectors of the different heads are placed in query, key, and value matrix respectively. This is done by multiplying the word embeddings with three matrices that are being trained during the training process. Let $\boldsymbol{W}^i$ denote the weight matrix used to obtain vector $\boldsymbol{i}$ where i is the query ($\boldsymbol{Q}$), key ($\boldsymbol{K}$), or value ($\boldsymbol{V}$) matrix. Consider an input sequence with the following dimensions (N, S, E) where N is the number of samples, S is the sequence length, and E is the embedding size (i.e. $E = d_{model}$). Let $x^i_{j,k} \in \mathbb{R}$ denote the k^{th} value of the embedding of the j^{th} token in the i^{th} sample for $i = 1, \cdots, N$, $j = 1, \cdots, S$, and $k = 1, \cdots, E$. The embeddings are depicted in Figure C.1.

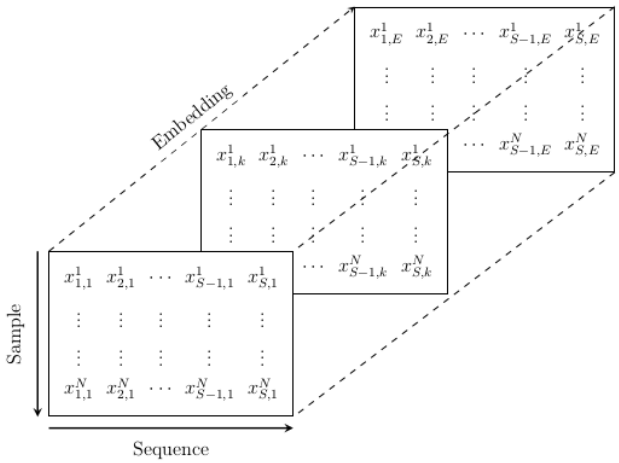


Figure C.1: Embedding of the input data.

For simplicity, consider the i^{th} sample (Figure C.2).

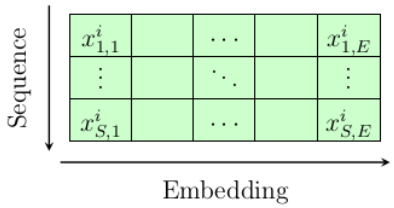


Figure C.2: ith sample of the input data.

Three separate linear layers are used to generate the query, key, and value. Each linear layer has its own weight matrix and bias vector. In each linear layer, the same bias vector is added to every column of the matrix resulting from the multiplication of the weight matrix and the input matrix. The input matrix is passed through these linear layers to obtain $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ (Figure C.3).

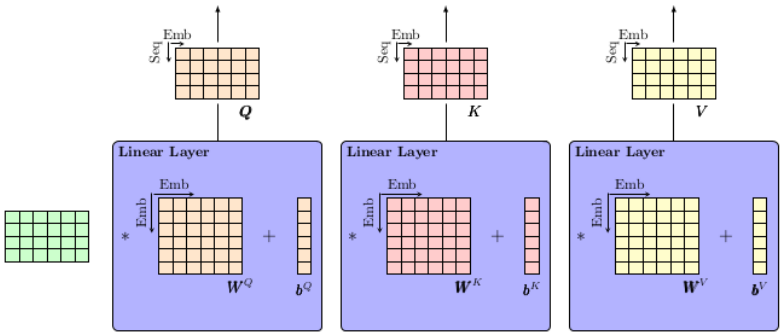


Figure C.3: Linear Layers used to generate the Query, Key, and Value matrices.

The computation of all heads can be performed using one matrix rather than requiring h matrices. This improves efficiency and simplifies the model. For example, the matrix $\boldsymbol{W}^Q$ is decomposed into h matrices of dimension $\mathbb{R}^{d_{model} \times d_{querysize}}$ where $d_{querysize} = d_{model} \cdot h^{-1}$, i.e.

$$\boldsymbol{W}^Q = \left[\boldsymbol{W}_1^Q; \cdots; \boldsymbol{W}_h^Q\right]$$

where $\boldsymbol{W}_i^Q \in \mathbb{R}^{d_{model} \times d_{querysize}}$ for $i = 1, \cdots, h$. This is depicted in Figure C.4.

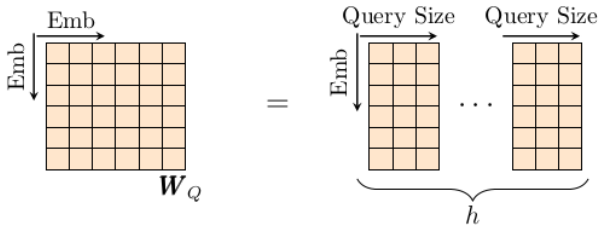


Figure C.4: Matrix $\boldsymbol{W}_Q$ decomposed into the h heads.

The matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ outputted by the three linear layers are reshaped to account for a head dimension. As a result of this reshaping, each query size slice corresponds to a matrix per head. The reshaping of $\boldsymbol{Q}$ is depicted in Figures C.5 and C.6.

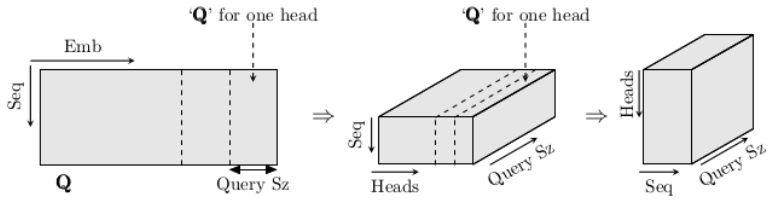


Figure C.5: Reshaping the matrix $\boldsymbol{Q}$ to include a head dimension. The matrix is reshaped again for ease of computation.

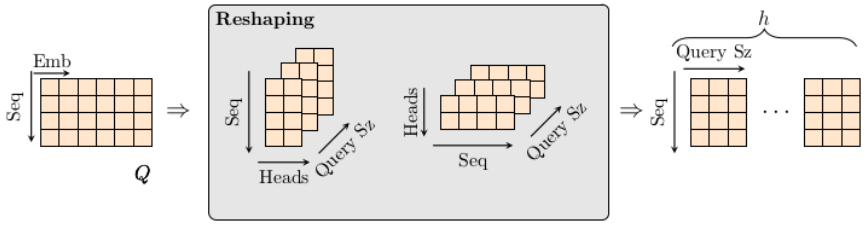


Figure C.6: Reshaping the matrix $\boldsymbol{Q}$ to include a head dimension. The matrix is reshaped again for ease of computation.

Once the three matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are split across the heads, the attention score for each head is computed. Let's consider head j. The first step is to multiply $\boldsymbol{Q}_j$ and $\boldsymbol{K}_j^T$.

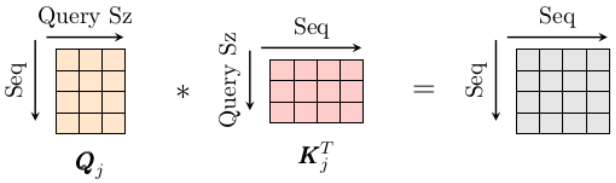


Figure C.7: Multiplication of the query and key matrices of the j^{th} head.

An optional mask is then added to the output of the multiplication. The mask is used to filter out unwanted entries when the *softmax* function is applied.

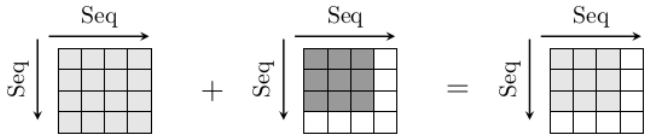


Figure C.8: Mask applied to the output of the multiplication.

The matrix of attention scores is then computed and all the attention scores from the h heads are merged together.

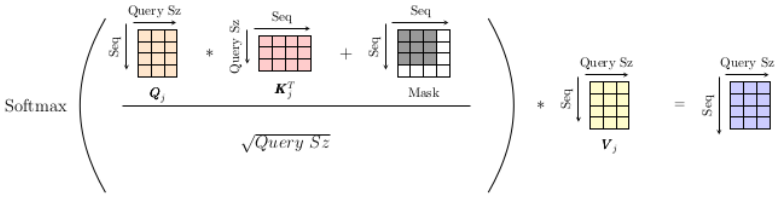


Figure C.9: Computation of the attention scores.

To merge the attention heads, the matrices are reshaped multiple times. This is depicted in Figure C.10.

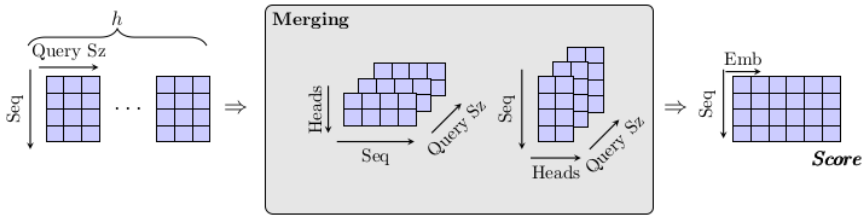


Figure C.10: Merging the scores.

Appendix D

Support Vector Machine

A Support Vector Machine (SVM) is a binary classifier. Consider the training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1,\cdots,N}$ where $\boldsymbol{x}_i \in R^I$ and $y_i \in \{-1, 1\}$. Define the hyperplane

$$\{\boldsymbol{x}: f(\boldsymbol{x}) \coloneqq \boldsymbol{x}^T\boldsymbol{\beta} + \beta_0 = 0\} \tag{D.1}$$

where $\boldsymbol{\beta} \in \mathbb{R}^I$ is a unit vector. A classification rule is defined as

$$G(\boldsymbol{x}) \coloneqq \text{sign}\left(f(\boldsymbol{x})\right) = \text{sign}\left(\boldsymbol{x}^T\boldsymbol{\beta} + \beta_0\right).$$

$f(\boldsymbol{x})$ (defined in Eq. (D.1)) computes the signed distance of a point $\boldsymbol{x}$ to the hyperplane $\boldsymbol{x}^T\boldsymbol{\beta} + \beta_0 = 0$. If the classes are separable, there exists a function $f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{\beta} + \boldsymbol{\beta}_0$ such that $y_i f(\boldsymbol{x}_i) > 0$ for all $i = 1, \cdots, N$. Let M be the distance between the hyperplane and the closest instance in either class, i.e. if $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$ are the closest points to the hyperplane such that $y_j = 1$ and $y_k = -1$, then $|f(\boldsymbol{x}_j)| = |f(\boldsymbol{x}_k)| = M$. $2M$ is called the margin (Figure D.1).

The optimisation problem an SVM solves is finding the hyperplane that creates the largest margin. This is formulated as

$$\begin{aligned} &\max_{\boldsymbol{\beta}, \beta_0 : \|\boldsymbol{\beta}\| = 1} \quad M \\ &\text{subject to} \quad y_i\left(\boldsymbol{x}_i^T\boldsymbol{\beta} + \beta_0\right) \geq M, \; i = 1, \cdots, N. \end{aligned}$$

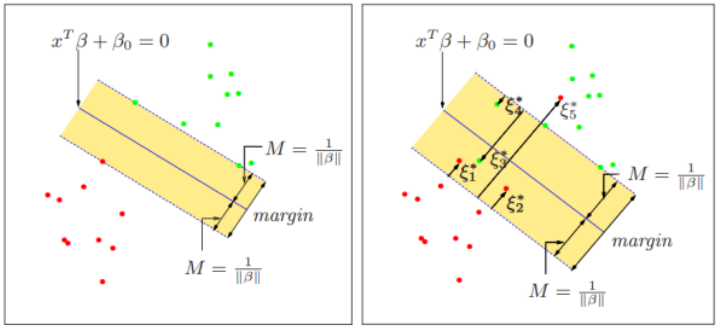


Figure D.1: SVM classifier. The left-hand side illustrates the separable case. The righ-hand side shows the non-separable case (Hastie et al. [13, Chapter 12 *Support Vector Machines and Flexible Discriminants*, Figure 12.1, page 418]).

If we define $M = \frac{1}{\|\boldsymbol{\beta}\|}$, the optimisation problem can be formulated as

$$\begin{aligned} \min_{\boldsymbol{\beta},\beta_0} \quad & \|\boldsymbol{\beta}\| \\ \text{subject to} \quad & y_i\left(\boldsymbol{x}_i^T\boldsymbol{\beta} + \beta_0\right) \geq 1,\ i = 1, \cdots, N. \end{aligned}$$

So far we have only considered the simple case when the data points are separable. This is often not the case. A common way of handling overlapping data points is allowing some points to slack and be on the wrong side of the margin. Define the slack variables $\boldsymbol{\xi} := (\xi_1, \cdots, \xi_N) \in \mathbb{R}^N$. The constraint is now modified to

$$y_i\left(\boldsymbol{x}_i^T\boldsymbol{\beta} + \beta_0\right) \geq M\left(1 - \xi_i\right) \tag{D.2}$$

such that $\xi_i \geq 0$ for $i = 1, \cdots, N$ and $\sum_{i=1}^{N} \xi_i \leq K$ where K is a constant. The value of ξ in the constraint presented in Eq. (D.2) is the proportional amount by which the prediction $f\left(\boldsymbol{x}_i\right)$ is on the incorrect side of the margin. By bounding the summation of the ξ_i's, we bound the total proportional amount by which predictions occur in the incorrect side of the margin. A misclassification occurs when $\xi_i > 1$ and therefore bounding the summation by K limits the number of misclassifications to K. The optimisation problem becomes

$$\min_{\boldsymbol{\beta},\beta_0} \quad \|\boldsymbol{\beta}\|$$

$$\text{subject to} \quad \begin{cases} y_i \left(\boldsymbol{x}_i^T \boldsymbol{\beta} + \beta_0 \right) \geq 1 - \xi_i \;\; i = 1, \cdots, N \\ \xi_i \geq 0, \; \sum_{i=1}^{N} \xi_i \leq K \end{cases} .$$

A more detailed description of SVMs is provided in Hastie et al. [13, Chapter 12 *Support Vector Machines and Flexible Discriminants*, pages 417-458].

Bibliography

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization, 2016. Preprint arXiv:1607.06450.

[2] Y. Bengio. Practical Recommendations for Gradient-Based Training of Deep Architectures. In *Neural Networks: Tricks of the Trade*, volume 7700, pages 437–478. Springer, 2012.

[3] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[4] M. Braverman, X. Chen, S. Kakade, K. Narasimhan, C. Zhang, and Y. Zhang. Calibration, entropy rates, and memory in language models. In *37th International Conference on Machine Learning, ICML 2020*, 37th International Conference on Machine Learning, ICML 2020, pages 1066–1076. International Machine Learning Society (IMLS), 2020.

[5] G. Chen. A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation, 2016. Preprint arXiv:1610.02583.

[6] U. Şimşekli, L. Sagun, and M. Gürbüzbalaban. A Tail-Index Analysis of Stochastic Gradient Noise in Deep Neural Networks. In *In Proc. Int'l Conf. Machine Learning*, 2019.

[7] A. M. Dai and Q. V. Le. Semi-supervised Sequence Learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[9] K. Doshi. Transformers Explained Visually (Part 3): Multi-head Attention, deep dive, 2017. Available at `https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853`.

[10] J. Eisenstein. *Introduction to natural language processing*. The MIT Press, 2019.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2017. Preprint arXiv:1706.02677.

[13] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2 edition, 2009.

[14] K. He and J. Sun. Convolutional neural networks at constrained time cost. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5353–5360, 2015.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[16] D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus), 2020. Preprint arXiv:1606.08415.

[17] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[18] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456, 2015.

[19] O. Irsoy and C. Cardie. Bidirectional Recursive Neural Networks for Token-Level Labeling with Structure. *CoRR*, abs/1312.0493, 2013.

[20] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[21] N. S. Keskar and R. Socher. Improving Generalization Performance by Switching from Adam to SGD. *CoRR*, abs/1712.07628, 2017.

[22] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[23] D. Krueger, T. Maharaj, J. Kramár, M. Pezeshki, N. Ballas, N. R. Ke, A. Goyal, Y. Bengio, A. C. Courville, and C. J. Pal. Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[24] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[25] I. Loshchilov and F. Hutter. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[26] B. Marr. What Is Unstructured Data And Why Is It So Important To Businesses? An Easy Explanation For Anyone, 2019. Available at

https://www.forbes.com/sites/bernardmarr/2019/10/16/what-is-unstructured-data-and-why-is-it-so-important-to-businesses-an-easy-explanation-for-anyone/?sh=7934163715f6.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[28] C. Olah. Understanding LSTM Networks, 2015. Available at https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[29] M. Pakkanen. Deep Learning Lecture Notes, 2020.

[30] J. Pennington, R. Socher, and C. D. Manning. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[31] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics, 2018.

[32] J. Pougué Biyong, B. Wang, T. Lyons, and A. Nevado-Holgado. Information Extraction from Swedish Medical Prescriptions with Sig-Transformer Encoder. *ACL Anthology*, pages 41–54, 2020.

[33] A. Radford and K. Narasimhan. Improving Language Understanding by Generative Pre-Training, 2018. Technical Report, OpenAI. URL: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[34] H. Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227–244, 2000.

[35] B. Singh, S. De, Y. Zhang, T. Goldstein, and G. Taylor. Layer-Specific Adaptive Learning Rates for Deep Networks. *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 364 – 368, 2015.

[36] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[37] W. L. Taylor. "Cloze Procedure": A New Tool For Measuring Readability. *Journalism Bulletin*, 30(4):415–433, 1953.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[39] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808. Association for Computational Linguistics, 2019.

[40] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4151–4161. Curran Associates Inc., 2017.

[41] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016. Preprint arXiv:1609.08144.

[42] K. You, M. Long, J. Wang, and M. I. Jordan. How Does Learning Rate Decay Help Modern Neural Networks?, 2019. Preprint arXiv:1908.01878.

[43] M. Zhang and Z. Zhou. A Review on Multi-Label Learning Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):1819–1837, 2014.

[44] P. Zhou, J. Feng, C. Ma, C. Xiong, S. Hoi, and W. E. Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning. In *34th Conference on Neural Information Processing Systems*, NeurIPS 2020, 2020.

Alden_Andrew_01666104

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63

PAGE 64

PAGE 65

PAGE 66

PAGE 67

PAGE 68

PAGE 69

PAGE 70

PAGE 71

PAGE 72

PAGE 73

PAGE 74

PAGE 75

PAGE 76

PAGE 77

PAGE 78