Applying Self-Attention Neural Networks for Sentiment Analysis Classification and Time-Series Regression Tasks

by

Artaches Ambartsoumian

B.Sc., Simon Fraser University, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Science

© Artaches Ambartsoumian 2018
SIMON FRASER UNIVERSITY
Fall 2018

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Approval

Name:	**Artaches Ambartsoumian**
Degree:	**Master of Science (Computing Science)**
Title:	**Applying Self-Attention Neural Networks for Sentiment Analysis Classification and Time-Series Regression Tasks**
Examining Committee:	**Chair:** Angelica Lim Assistant Professor **Fred Popowich** Senior Supervisor Professor **Anoop Sarkar** Supervisor Professor **Jiannan Wang** External Examiner Assistant Professor School of Computing Science
Date Defended:	**November 26, 2018**

Abstract

Many machine learning tasks are structured as sequence modeling problems, predominantly dealing with text and data with a time dimension. It is thus very important to have a model that is good at capturing both short range and long range dependencies across sequence steps. Many approaches have been used over the past few decades, with various neural network architectures becoming the standard in recent years. The main neural network architecture types that have been applied are recurrent neural networks (RNNs) and convolutional neural neworks (CNNs). In this work, we explore a new type of neural network architecture, self-attention networks (SANs), by testing on sequence modeling tasks of sentiment analysis classification and time-series regression. First we perform a detailed comparison between simple SANs, RNNs, and CNNs on six sentiment analysis datasets, where we demonstrate SANs achieving higher classification accuracy while having other better model characteristics over RNNs such as faster training and inference times, lower number of trainable parameters, and consuming less memory during training. Next we propose a more complex self-attention based architecture called ESSAN and use it to achieve state-of-the-art (SOTA) results on the Stanford Sentiment Treebank fine-grained sentiment analysis dataset. Finally, we apply our ESSAN architectures for the regression task of multivariate time-series prediction. Our preliminary results show that ESSAN once again achieves SOTA results, beating previous SOTA RNN with attention architectures.

Keywords: Neural Networks, Self-Attention Networks, Sequence Modeling, Natural Language Processing, Sentiment Analysis, Time Series Prediction

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my senior supervisor, Dr. Fred Popowich, for all of his support throughout my degree. I am grateful for the freedom he patiently provided to me to pursue all of my research interests, as well as the perfect amount of guidance that kept me moving towards completing this thesis. I would also like to thank my supervisor, Dr. Anoop Sarkar, and my thesis examiner, Jiannan Wang, for their valuable questions and feedback. Thanks also to Angelica Lim for chairing my thesis defence.

Last but not least, I would like to thank my family and my partner, Daphne. Their constant support, love, and encouragement helped me immensely throughout my studies.

Table of Contents

Approval ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures ix

1 Introduction 1

1.1 Contribution . . . 2
1.2 Overview . . . 3

2 Background 5

2.1 Neural Networks Fundamentals . . . 5
2.2 Feed Forward Neural Networks . . . 6
2.3 Recurrent Neural Networks, Gated Cells, Bidirectional RNN . . . 8
2.4 Encoder-Decoder Architecture, Attention Mechanism . . . 11
2.5 CNNs as Alternatives to RNNs . . . 14

3 Self-Attention Networks 16

3.1 Self-Attention Mechanism . . . 16
3.2 Transformer Architecture . . . 17
3.2.1 Motivation . . . 17
3.2.2 Architecture . . . 18
3.3 Position Information Techniques . . . 20

4 Proposed Architectures & Implementation Details 23

4.1 Simple Self-Attention Network (SSAN) . . . 23
4.2 Extended SSAN (ESSAN) . . . 24
4.2.1 Masking . . . 25

5 Sentiment Analysis **28**
5.1 Comparison of Building Blocks: SAN vs RNN vs CNN 29
5.1.1 Experiments 29
5.1.2 Datasets Used 29
5.1.3 Pre-Trained Word Embeddings 31
5.1.4 Baselines Descriptions 31
5.1.5 Self-Attention Architectures 32
5.1.6 Analysis 32
5.1.7 Model Characteristics 33
5.1.8 Discussion 35
5.2 SOTA Approach for SST Datasets 35
5.2.1 SST Dataset Processing 36
5.2.2 Experiments 36
5.2.3 Results & Discussion 37
5.2.4 Model Characteristics 39

6 Time-Series Regression **40**
6.1 Dataset Description 40
6.2 Baselines 41
6.3 Experiments 42
6.4 Discussion 43

7 Conclusion **45**
7.1 Summary 45
7.2 Future Work 45

Bibliography **47**

List of Tables

Table 2.1 Shows the relationship between the number of sequence steps fed to the input layer and the number of trainable weights for the first hidden layer. Each sequence step is a 300 dimensional vector, the hidden layer dimension size is also 300. 8

Table 2.2 Equations for calculating the hidden state of LSTM and GRU cells for sequence position t. $\otimes$ is the element-wise product operation, and σ is the sigmoid function. 10

Table 3.1 from [Vaswani et al., 2017]: "Shows the maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, and k is the kernel size of convolutions." 18

Table 5.1 Details of hardware and software that were used for all experiments. 30

Table 5.2 Modified Table 2 from [Barnes et al., 2017]. Dataset statistics, embedding coverage of dataset vocabularies, as well as splits for Train, Dev (Development), and Test sets. The 'Wiki' embeddings are the 50, 100, 200, and 600 dimension used for experiments. 31

Table 5.3 Modified Table 3 from [Barnes et al., 2017]. The baseline results are taken from [Barnes et al., 2017]; the self-attention models results are ours. Test accuracy averages and standard deviations (in brackets) of 5 runs. Best model for each dataset is given in **bold**. 34

Table 5.4 Neural networks architecture characteristics. A comparison of number of learnable parameters, GPU VRAM usage (in megabytes) during training, as well as training and inference times (in seconds). 35

Table 5.5 Hyper-parameters used by ESSAN for SST-fine and SST-binary datasets. . . 37

Table 5.6 Test accuracy of fine-grained sentiment analysis on SST datasets. The "Top" columns contain the highest accuracy that each method reported. The "Avg" columns report mean and standard deviation (in bracket) of 5 runs. 38

Table 5.7 ESSAN vs DiSAN comparison of number of learnable parameters, GPU VRAM usage (in megabytes) during training, as well as training and inference times (in seconds). 39

Table 6.1 The driving series information for number of driving series and train/validation/test splits for the SML 2010 dataset. 40
Table 6.2 MAE, MAPE, and RMSE metric equations. 42
Table 6.3 Hyper-parameters used by ESSAN for SML2010 dataset. 43
Table 6.4 Time series prediction metric results for the SML2010 dataset. The results are mean and standard deviation of 10 runs for the three metrics: MAE, MAPE, and RMSE. The best results for each metric are outlined in **bold**. For each $model(d_{model})$, the dimension of the encoder m and the decoder p is set to $m = p = d_{model} =$ 64 or 128. 44

List of Figures

Figure 2.1 A Single Neuron. 5

Figure 2.2 Feed-forward neural network example of processing a sequence. 7

Figure 2.3 Recurrent neural network example of processing a sequence. 9

Figure 2.4 Modified Figure 1 from [Sutskever et al., 2014]. Visualizes a seq2seq model processing input "ABC" to produce output "WXYZ". The encoder $LSTM1$ network processes the input sequence to create a sequence representation vector. The decoder $LSTM2$ network uses the encoder's hidden state vector as input to create an output sequence. 12

Figure 2.5 Figure 1 from [Bahdanau et al., 2014]. Visualizes the process of how a bidirectional encoder-decoder architecture with attention generates the t-th target sequence item from input sequence $(x_1, x_2, ..., x_T)$. 13

Figure 2.6 Figure 1 from [Kalchbrenner et al., 2016]. The architecture of ByteNet. A demonstration of the process of translating from source (in red) to target language (in blue). 1-D dilated convolutions are used for source sentence encoding. 15

Figure 3.1 Figure 1 from [Vaswani et al., 2017]. Visualization of the Transformer model architecture computation graph. 19

Figure 3.2 Figure 1 from [Shaw et al., 2018]. An example illustrating some of the edge values that represent relative positions between sequence positions. A clipping distance of k is chosen, where $2 <= k <= n - 4$ is assumed. . . 22

Figure 4.1 SSAN Model Architecture 24

Figure 4.2 Masking sequence after feed-forward layers example. 26

Chapter 1

Introduction

Many machine learning problems perform sequence modeling because much of the real world data is structured in a sequential manner. Common examples are any data that have a time component (e.g. weather data, stock prices, audio signals), DNA sequences, as well as natural language in a form of text. Many natural language processing (NLP) problems are structured to perform sequence modeling. Sequence modeling is done for both classification and regression tasks, most commonly for modeling sequential input data but sometimes also for generating a sequence as output.

The most common classification and regression problems involve having a single variable for output. Such problems require the model used to learn a mapping function f from an input sequence $X = (x_1, x_2, ..., x_t)$ to a variable y, where t is the input sequence length. In the case of classification, $y \in \{c_1, c_2, ..., c_n\}$ is a discrete variable representing classes/categories, where n is the number of possible classes. Some examples of such classification problems are sentiment analysis, spam detection, and topic classification. In the case of regression, $C \in \mathbb{R}$ is a continuous variable. Some examples of such regression problems are price prediction of assets like stocks or commodities, and weather temperature forecasting. Tasks that require sequences for output also utilize sequence modeling. In the case of image captioning, sequence modeling is only utilized for producing the output textual caption and not the input image. Other tasks like machine translation utilize sequence modeling for both processing the input source language sentence as well as generating the output target language sentence, such problems are also known as "sequence to sequence" tasks.

An important attribute of sequence modeling is the ability to capture short-range and long-range dependencies, that is model/capture relationships between sequence positions. Models need to be good at capturing such dependencies in order to perform well when processing long sequences. This is something many sequence modeling approaches struggle with [Bengio et al., 1994, Hochreiter, 2001], and thus is often the motivation behind newer models that have been proposed over the years [Hochreiter and Schmidhuber, 1997, Bahdanau et al., 2014, Vaswani et al., 2017]. Modeling such dependencies is important for both time-series prediction and NLP tasks like sentiment analysis. Here is an example that demonstrates long and short range dependencies: "The movie plot line was a bit dull, the theatre seats were uncomfortable, food choice was limited, but it had great visual effects which made the overall experience fantastic." In this relatively long sentence, a short-range

dependency is that 'uncomfortable' refers to 'seats'. A long-range dependency would be 'had great visual effects' referring to 'the movie' at the beginning of the sentence. Both of these dependencies are difficult to capture for many of the current neural network methods. Generally, the long-range dependencies are the more difficult to capture. However, modeling short-range dependencies can also be difficult, especially if they are mentioned in the beginning of a long sentence. This is because models such as basic RNNs process the input sequentially, which causes them remember only the last few words in the sentence.

Self-attention networks (SANs) are a new type of neural network architecture that have been shown to be effective for various sequence to sequence problems, where capturing short and long-range dependencies is important [Vaswani et al., 2017, Liu et al., 2018]. In this thesis, we focus on applying SAN architectures for supervised sequence modeling problems. We explore various SAN architectures and compare their attributes to previous RNN and CNN based neural network architectures. We test on the two most popular types of supervised tasks, classification and regression. We chose sentiment analysis for the classification task and time-series prediction for the regression task. We chose these two problems because they both involve sequence modeling, extensive research has been done for them with the current main approaches being neural network methods, as well as our personal preference. Sentiment analysis is the process of determining the sentiment of a piece of text, e.g. positive, negative, or neutral [Turney, 2002, Pang and Lee, 2008]. Time-series prediction is the task of determining the future numerical value of interest based on historical values [Weigend, 2018], e.g. predicting the price of a company's stock tomorrow based on the prices for the past few weeks. If multiple time series features are used for input, the task is called multivariate time-series prediction [Chakraborty et al., 1992, Qin et al., 2017].

1.1 Contribution

In this thesis, we propose and apply various self-attention networks for sentiment analysis and time-series prediction. Our contributions are as follows:

- We compare how the self-attention building block performs compared to recurrence and convolution. We demonstrate that our simple self-attention network (SSAN) architecture outperforms simple recurrent networks (LSTM and BiLSTM) as well as simple CNN architecture on six sentiment analysis datasets.
- We also demonstrate that self-attention networks have other better characteristics such as lower memory consumption, lower number of trainable parameters, and faster training/inference times than RNNs.
- Our proposed ESSAN architecture achieves state-of-the-art (SOTA) accuracy on the SST-fine dataset, all while having better model characteristics (training and inference speed as well as memory usage) compared to DiSAN[Shen et al., 2018a], the previous SOTA SAN architecture.

- Finally, we apply our ESSAN architecture for the time-series prediction regression task, and show SOTA performance on one dataset, beating the previous SOTA RNN with attention approaches.

1.2 Overview

In Chapter 2, we will cover the previous common approaches that were used for sequence modeling problems. Primarily, we will discuss in depth the inner-workings of neural networks architectures such as feed-forward neural networks (FFNNs), Recurrent Neural Networks (RNNs), RNNs with attention mechanism, and Convolution Neural Networks (CNNs).

In Chapter 3, the Transformer [Vaswani et al., 2017] architecture and the motivation behind it will be covered due to its influence on this research. The self-attention mechanism will be covered in depth as well as the various ways of including input sequence positional information into SANs.

In Chapter 4, we will cover our proposed SAN architectures, the SSAN and ESSAN, that will be used in the application Chapters 5 and 6.

Chapter 5 is split into two main parts. The first part covers our work done for [Ambartsoumian and Popowich, 2018], where we demonstrate the benefits of the self-attention mechanism over the recurrence and convolution counter-parts. In our paper, we extend the work of [Barnes et al., 2017] by comparing our SSAN architecture to their LSTM, BiLSTM and CNN architectures on the exact same six datasets. To minimize any process deviations, we use their open-source codebase and only change the model as well as some minor batch-feeding methodology. We re-use their code for datasets pre-processing, accuracy metric evaluation, as well as use the exact same word embeddings. We demonstrate that SAN models have generally better accuracy across all six datasets than the RNN and CNN models. Next, for all architectures we explore other model characteristics such as memory consumption during training, number of trainable parameters, and train and inference times. Finally, in this section we also explore various SAN architecture variations, specifically the methods of incorporating sequence position information. We show that using Relative Position Representations [Shaw et al., 2018] method performs the best for multiple SAN architectures on the same six datasets, while having negligible effect on number of trainable parameters as well as training and inference efficiency.

In the second part of Chapter 5, we apply our ESSAN architecture to achieving state-of-the-art (SOTA) results on the SST-fine dataset, one of the more competitive sentiment analysis datasets. For this work, we achieve better accuracy on the SST-fine dataset than the previous SOTA self-attention network, DiSAN [Shen et al., 2018a], while having half the trainable parameters, consuming four times less memory and having faster train and inference times.

In Chapter 6, we apply our ESSAN architectures on a completely different task, time-series prediction. Here we show that SANs are also effective for regression problems that involve sequence modeling. We use the SML2010 dataset from [Qin et al., 2017] and compare our results to their

models. We show an improvement over their Dual-Stage Recurrent Neural Network (DA-RNN) that achieved the previous best result on this dataset.

Finally, in Chapter 7, we summarize the contributions of this thesis and talk about possible future work.

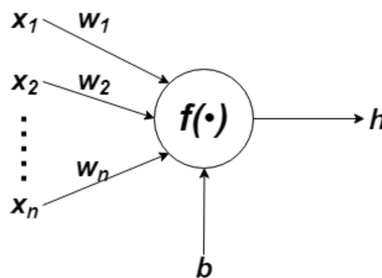
Chapter 2

Background

In order to understand the motivation for this thesis, first some core neural network concepts as well as history needs to be discussed. First, standard feed-forward neural networks will be introduced as the concepts for these networks are the foundation for all neural network architectures. Next, we'll discuss the most common variant of neural networks used for sequence modeling, Recurrent Neural Networks (RNNs) and its many variations. Finally, we'll briefly cover Convolution Neural Networks (CNNs), another popular architecture that traditionally has been used for image processing tasks, but lately has been applied to sequence modeling in order to overcome some of RNN's shortcomings. Parts of this chapter are taken from our work for [Ambartsoumian and Popowich, 2018].

2.1 Neural Networks Fundamentals

Figure 2.1: A Single Neuron.



The foundation of neural networks is the concept of an artificial neuron, also called a perceptron, with primary work dating back to [McCulloch and Pitts, 1943] and [Rosenblatt, 1958]. The neuron is the building block of all neural network architectures. The operation of a neuron is simple and can be described as follows: multiply all the input variable values by unique weights, then sum the outputs of the previous step and add an optional bias value, finally apply a non-linear activation function to create what is called a hidden state. Given a set of input features $X = (x_1, x_2, ..., x_n)$, a set of weights $W = (w_1, w_2, ..., w_n)$, a bias value b, and an activation function f. The process of

calculating a hidden state h can be summarized by the following equations:

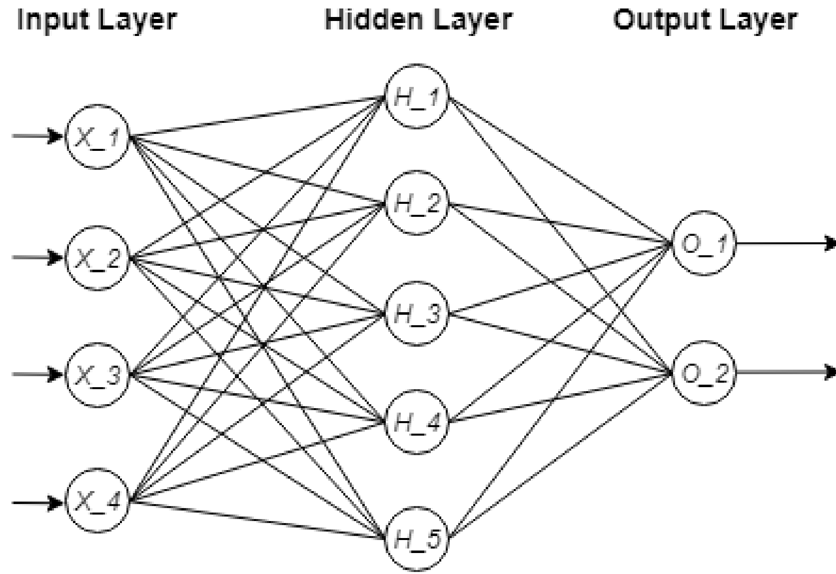
$$h = f(b + \sum_{i=1}^{n} x_i * w_i) \tag{2.1}$$

A neuron outputting a high value is regarded as the neuron "firing", which means it has detected a pattern among the input that it's trained to recognize. The output of one neuron is often the input for another, which creates a network of neurons, hence "neural networks". The latter neurons will be firing if a desired pattern among previous neurons is found, a process of detecting patterns in patterns. The goal is to find values for weights w for each neuron, and the bias b, such that the final output of the network, $\hat{Y}$, is as close as possible to the expected output, Y, for the task at hand. The most common methods for optimizing a neural network's trainable parameters is called gradient descent. For the machine learning problem of interest, a differentiable cost function, also known as loss function, $C(\hat{Y}, Y)$ needs to be defined that measures how incorrect the output $\hat{Y}$ of the network is compared to expected output Y. For each training iteration on some training data, each weight is updated as $w_i^{new} = w_i - \eta * \Delta w_i$, where $\Delta w_i = \frac{\partial w_i}{\partial C(\hat{Y}, Y)}$ is a partial derivative calculating how much the weight contributed to the final cost, and η is the learning rate. An efficient algorithm for calculating the partial derivatives for all weights in the network is back-propagation [Rumelhart et al., 1986]. Back-propagation and its variants is the most common approach for computing weight gradients for training.

2.2 Feed Forward Neural Networks

The simplest neural network is the feed-forward neural network (FFNN), also known as the multi-layer perceptron. The input is fed into layers of neurons, often stacked layers where the output of one layer is propagated to be the input of the next. An example of such a network is shown in Figure 2.2. This example shows 4 input features, (X_1, X_2, X_3, X_4), fed into a hidden layer of five neurons, $H = (H_1, H_2, H_3, H_4, H_5)$, outputs of which are then fed into two output units (O_1, O_2). To keep things simple, each X_i input demonstrates a single value, though in practice it is usually a vector of numbers. The lines connecting the units are the weights, also referred to as synapses. The output units are the same as neurons, except typically no non-linear activation function is applied. All parts of this simple FFNN can be customized, such as the amount of input units, the number of hidden layers, the number of neurons in each hidden layer, as well as the number of output units. This customizability makes FFNNs a highly versatile model, capable of being applied to many machine learning problems.

Figure 2.2: Feed-forward neural network example of processing a sequence.



The key for this architecture is that the input size is fixed and that information flows only in one direction - that is, output of each layer is solely determined from the output of the previous layer. This constraint makes inference and training calculations easy to implement by using basic linear algebra operations. The hidden layer from the example in Figure 2.2 can be calculated using the following formula:

$$H = f(b + X * W) \tag{2.2}$$

where $X \in \mathbb{R}^{1\times 4}$, $W \in \mathbb{R}^{4\times 5}$, and $H \in \mathbb{R}^{1\times 5}$. The weighted sum operations for each hidden layer neuron can computed with a single matrix multiply. Running such matrix and vector operations is extremely fast on graphics processing units (GPUs).

If we let the input units X from Figure 2.2 be values for sequence steps of a sequence, the FFNN can be applied for sequence modeling problems. However, there are major limitations when using FFNNs for sequence modeling. First, the number of input feature units must be fixed as we train the network. This forces us to make the input layer contain as many units as the largest sequence in the training set requires. In practice it's difficult to scale FFNNs to longer sequences, as the number of weights for the first layer grows linearly with respect to the number of input sequence steps fed. Second, the neural network is not aware of the order of the data. The input sequence can be shuffled, which will not change the training behaviour of the network.

To demonstrate that FFNNs are impractical for long sequences, we provide a realistic example. Suppose our sequence modeling problem contains 300 features for each sequence step and we set the first hidden layer to be 300 neurons. Then the total number of weights for the first hidden layer will be $N = 300 * T * 300$, where T is the number of sequence steps we feed into the input layer. We are essentially adding $300 * 300$ more weights/synapses in order to be able to model sequences that are

one length longer. Table 2.1 shows this relationship. In our experience, an average sentence length for various NLP problems is around 20 to 30 words, this would result in the first layer having between $1,800,000$ and $2,700,000$ learnable weights. However, we can't go by average sentence length, we need to our model to be able to process the longest sentence in the dataset. In our experience, the longest sentences for many most datasets range between 60 to 120 words, requiring $5,400,000$ to $10,800,000$ weights. The choices for input dimension size per sequence-step and the hidden layer size are reasonable, in fact we use these values for our SAN models for the sentiment analysis task. However, five to ten million parameters is an absurd amount to simply process a sequence. The results will be a model that will be slow to train due to large matrix multiplication as well as it will require large amounts of memory. In order to cut down the number of weights, it is evident that reusing weights for processing sequence steps is required, which is the motivation for the creation of RNNs.

T	**N**
5	$450,000$
20	$1,800,000$
30	$2,700,000$
60	$5,400,000$
120	$10,800,000$

Table 2.1: Shows the relationship between the number of sequence steps fed to the input layer and the number of trainable weights for the first hidden layer. Each sequence step is a 300 dimensional vector, the hidden layer dimension size is also 300.

2.3 Recurrent Neural Networks, Gated Cells, Bidirectional RNN

Feed-forward neural networks are great function approximators that are commonly used when input data is of fixed size. In the previous section, we have showed that feed-forward networks don't scale to large input sequences, this is where Recurrent Neural Networks (RNNs) come in. RNNs were designed specifically for processing sequences, with initial work on being done in late 1980's [Werbos, 1988] and early 1990's [Elman, 1990, Werbos, 1990, Williams and Zipser]. For the past five years, RNN-based architectures have been the dominant approach for NLP tasks [Young et al., 2017], with CNNs and fully-attentional models gaining popularity only in the past couple of years.

RNNs reuse trainable weights across sequence positions by processing the input sequence one step at a time, for example one word of a sentence at a time. An RNN's hidden state is updated as each sequence step of the input is processed, unlike FFNN where the hidden state is calculated in a single forward pass for the whole input sequence. The overall structure of the network is similar to FFNNs, except that the hidden units are also conditioned on the previous states of the hidden layer from processing previous sequence positions. Using the sequence problem definition from previous

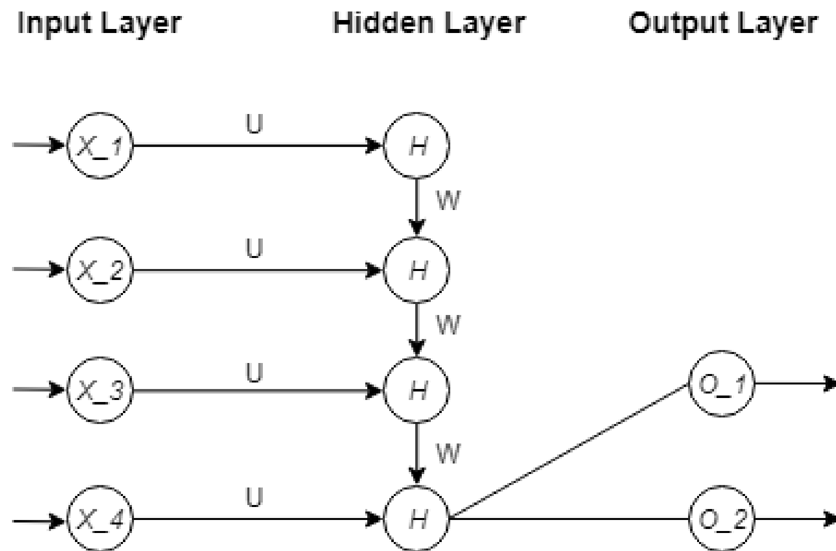
section, the hidden state of standard RNN is computed like so:

$$h_i = f(b + h_{i-1} * W + x_i * U) \tag{2.3}$$

where h_i is the hidden state after processing the input sequence to to position i, x_i is the data for sequence position i, $f($ is the non-linear activation function (typically $tanh$), learnable parameters are $b \in \mathbb{R}^N$, $W \in \mathbb{R}^{N \times M}$, and $U \in \mathbb{R}^{M \times M}$. Note that h_0 needs to be initialized to zeroes (since no data is yet processed). The standard RNN has two main sets of trainable parameters that are used for processing each sequence position, the weights U that are used for processing the input sequence position, and the weights W that are used for processing the previous hidden states. A demonstration of how the example from previous section can be adapted to this architecture is illustrated in Figure 2.3.

The sequential processing on the input sequence decouples the number of trainable parameters in the network from the input sequence length; this allows RNNs to be able to process sequences of arbitrary size. To accommodate for the hidden state being the product of processing varied length sequences, back-propagation through time (BPTT) [Werbos, 1990] was introduced as a variant of BP to calculate the gradients required for training.

Figure 2.3: Recurrent neural network example of processing a sequence.



As with FFNNs, there are some critical downsides to the standard RNN architecture, which inspired improved RNN architectures. The two main issues that affect standard RNNs in practice are the vanishing and exploding gradient problems [Bengio et al., 1994, Pascanu et al., 2013]. Much like very deep FFNNs, RNNs require long sequential matrix multiplications for their calculations. Processing a sequence of length 10 with an RNN is similar computationally to computing the forward pass for a 10 layer FFNN. In practice, calculating the gradients for one of the earlier weight multiplications operations via BP or BPTT can lead to gradients for these weights to be minuscule,

LSTM	GRU
$\mathrm{f}_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$ $\mathrm{i}_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$ $\mathrm{o}_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$ $\tilde{C}_t = \tanh(U_c x_t + W_c h_{t-1} + b_C)$ $\mathrm{C}_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t$ $\mathrm{h}_t = \tanh(C_t) \otimes o_t$	$\mathrm{z}_t = \sigma(U_z x_t + W_z h_{t-1} + b_z)$ $\mathrm{r}_t = \sigma(U_r x_t + W_r h_{t-1} + b_r)$ $\tilde{h}_t = \tanh(U_i x_t + W_i(r_t \otimes h_{t-1}) + b_{\tilde{h}_t})$ $\mathrm{h}_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes \tilde{h}_t$

Table 2.2: Equations for calculating the hidden state of LSTM and GRU cells for sequence position t. $\otimes$ is the element-wise product operation, and σ is the sigmoid function.

and thus no learning happens, or be very large, in which case training becomes unstable and doesn't converge. In the case of very large ("exploding") gradients, a simple solution is to simply clip to prevent them from getting to0 large [Pascanu et al., 2013]. The solutions for the vanishing gradient problem are more complex and are often tied with RNN-based architecture's ability to learn long-range dependencies. The most popular solution that significantly helps with the vanishing gradient problem is to use gated RNN cells such as LSTMs [Hochreiter and Schmidhuber, 1997] or GRUs [Cho et al., 2014a]. In fact, these gated cells are so popular that they've become synonyms with RNN as the architecture itself.

Gated RNN cell networks, like LSTMs and GRUs, allow the RNN's hidden state to not be updated when processing some sequence positions if the information is deemed not important to include in the hidden state. Such filtering ability allows these networks to be much better at modeling long-range dependencies. For example, if the first and last word of some sentence are the most important for some task, we would like the final hidden state of the RNN to reflect that. In practice, the hidden state of the standard RNN cell tends to incorporate the information from the most recently processed sequence items, thus the standard RNNs would struggle to keep the information of the first word of the sentence. Gated cells on the other hand have the ability let their hidden state be composed only of the first and last words of the sentence; and in practice these networks can model much longer sequences [Karpathy, 2015]. The equations for calculating the hidden state of LSTM and GRU cells are detailed in Table 2.2.

The LSTM cell has two states that it tracks, the cell's internal memory state C_t, and the output hidden state h_t. It has 3 gates f_t, i_t, o_t that are called the forget, input, and output gates respectively. They are called gates as they output values between 0 and 1 (due to the sigmoid activation function), which are then used in an element-wise multiplication operations to determine how much information should flow through the respective gates. $\tilde{C}_t$ is the candidate cell memory state, and it's computed in the exact same way as the hidden state for the standard RNN cell. The forget f_t and input i_t gates are used to determine how to combine the old cell memory state C_{t-1} with the new candidate cell memory $\tilde{C}_t$. Finally, the hidden state of the LSTM cell is determined by regulating how much of the cell memory state to output, using the output o_t gate. The hidden state h_t is different from the

internal memory state C_t to allow the cell to hold internal memory that may not yet be relevant for other parts of the network to see. For example, if there is a long-range dependency between the first and last word of a sentence, the cell state C_t could contain the information from the first word of the sentence and only let that information through to the hidden state h_t when processing the last word. This ability is what makes LSTMs much better at modeling long-range dependencies than the standard RNN [Karpathy, 2015].

The GRU cell is a simple version of the LSTM cell and has been found to perform comparably on various NLP tasks, while having fewer parameters[Chung et al., 2014]. The GRU cells doesn't have an internal memory state like the LSTM, and thus also doesn't need an output gate to decide what information to output.

The introduced gated cells are great at constructing hidden states that reflect the sequence positions that have been processed. However, what if we require the hidden state at step t to also reflect the subsequent positions? For example, processing a word in the middle of a sentence, it would be desirable to construct a hidden state vector that not only represent the current word and how it relates to all the previous words, but also how it relates to the rest of the words in the sentence. In such cases, a bi-directional RNN can be used [Schuster and Paliwal, 1997]. The idea is quite simple, we reverse the input sequence and use a second RNN network to process it, the hidden states of both RNNs is then concatenated. Formally the process can be described by the following steps:

1. Use the first RNN, such as an LSTM, network to process input sequence $(x_1, x_2, ..., x_T)$ to produce hidden states $(h_1, h_2, ..., h_T)$.

2. Use the second RNN network to process a reversed input sequence $(x_T, x_{T-1}, ..., x_1)$ to produce hidden states $(\hat{h}_T, \hat{h}_{T-1}, ..., \hat{h}_1)$.

3. Concatenate the hidden states, position-wise, to create final hidden states $([h_1, \hat{h}_T], [h_2, \hat{h}_{T-1}], ..., [h_T, \hat{h}_1])$.

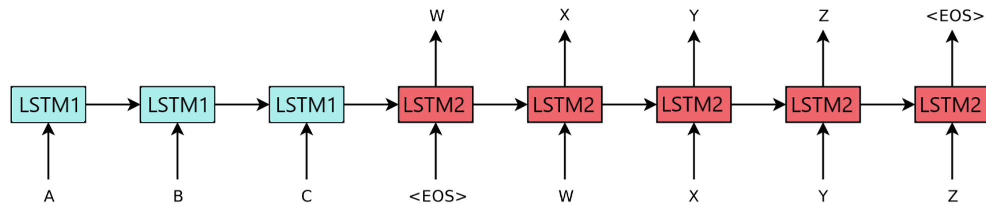
The resulting hidden states now have the ability to reflect how any sequence position relates to the entire sequence, and not just the previous positions. This approach has been successfully applied for many tasks such as sentiment analysis [Zhou et al., 2016], neural machine translation [Bahdanau et al., 2014], dependency parsing [Kiperwasser and Goldberg, 2016], and POS tagging [Plank et al., 2016].

2.4 Encoder-Decoder Architecture, Attention Mechanism

The most common use case of the RNN cells discussed is for the encoder-decoder architecture that is used for sequence to sequence (seq2seq) tasks [Cho et al., 2014a, Sutskever et al., 2014]. It is used for such tasks as Neural Machine Translation (NMT) [Wu et al., 2016, Bahdanau et al., 2014, Luong et al., 2015], text summarization [Nallapati et al., 2016, Rush et al., 2015], chatbots [Vinyals and Le, 2015], and time-series prediction [Qin et al., 2017]. In this architecture, one RNN cell encodes

the input sequence, and the second RNN cell is used for producing the desired output sequence; a visualization example of this process is provided in Figure 2.4.

Figure 2.4: Modified Figure 1 from [Sutskever et al., 2014]. Visualizes a seq2seq model processing input "ABC" to produce output "WXYZ". The encoder $LSTM1$ network processes the input sequence to create a sequence representation vector. The decoder $LSTM2$ network uses the encoder's hidden state vector as input to create an output sequence.

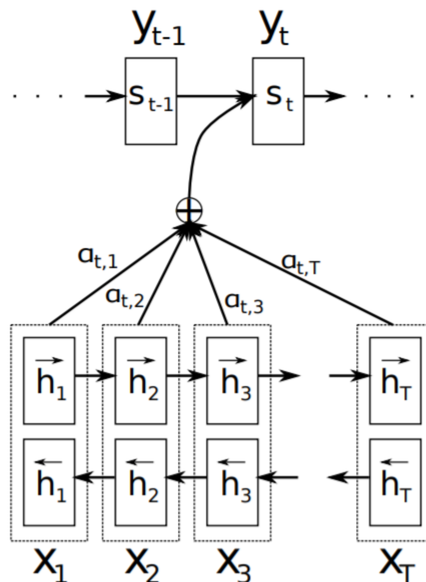


A critical detail of the Seq2Seq architecture is that the decoder is only given a single vector from the encoder. It is the encoder's job to create a fixed vector that incorporates all the information of the input sequence that the decoder will require; in practice this is a bottleneck. One could increase the hidden vector size to theoretically increase the expressive capacity of the hidden state, but there is a limit to how much that is practical. To improve the standard encoder-decoder architecture, a new mechanism was introduces, called attention, that allows the decoder to also consider the encoder hidden states at every single sequence position.

The attention mechanism was introduced by [Bahdanau et al., 2014] to improve the standard RNN encoder-decoder sequence-to-sequence architecture for NMT [Sutskever et al., 2014]. Since then, it has been extensively used to improve various RNN and CNN architectures [Cheng et al., 2016a, Kokkinos and Potamianos, 2017, Lu et al., 2016]. The attention mechanism has been an especially popular modification for RNN-based architectures due to its ability to improve the modeling of long range dependencies ([Daniluk et al., 2017, Zhou et al., 2018]).

Originally [Bahdanau et al., 2014] described attention as the process of computing a context vector for the next decoder step that contains the most relevant information from all of the encoder hidden states by performing a weighted sum on the encoder hidden states. How much each encoder state contributes to the weighted average is determined by an alignment score between that encoder state and previous hidden state of the decoder. A visualization of this process for a bidirectional encoder-decoder can be seen in Figure 2.5.

Figure 2.5: Figure 1 from [Bahdanau et al., 2014]. Visualizes the process of how a bidirectional encoder-decoder architecture with attention generates the t-th target sequence item from input sequence $(x_1, x_2, ..., x_T)$.



More generally, we can consider the previous decoder state as the query vector, and the encoder hidden states as key and value vectors. The output is a weighted average of the value vectors, where the weights are determined by the compatibility function between the query and the keys. Note that the keys and values can be different sets of vectors [Vaswani et al., 2017].

The above can be summarized by the following equations. Given a query q, values $(v_1, ..., v_n)$, and keys $(k_1, ..., k_n)$ we compute output z:

$$z = \sum_{j=1}^{n} \alpha_j (v_j) \tag{2.4}$$

$$\alpha_j = \frac{\exp score(k_j, q)}{\sum_{i=1}^{n} \exp score(k_i, q)} \tag{2.5}$$

α_j is computed using the softmax function where $score(k_i, q)$ is the compatibility score between k_i and q. Many compatibility functions, $score(k, q)$, have been proposed:

Name	Equation
Additive [Bahdanau et al., 2014]	$v_a^T tanh(W_a[k, q])$
General [Luong et al., 2015]	$k^T W_a q$
Dot-Product [Luong et al., 2015]	$(k)^T (q)$
Scaled Dot-Product [Vaswani et al., 2017]	$\frac{(k)^T (q)}{\sqrt{d_k}}$

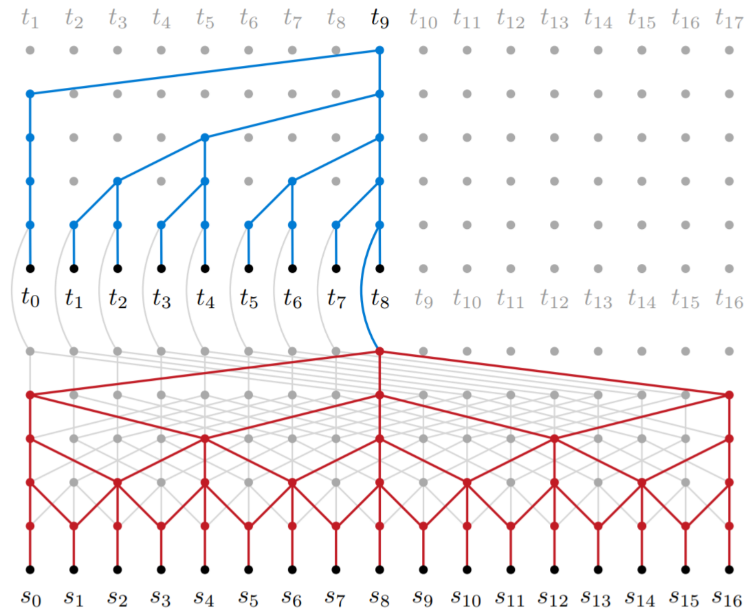
where v_a^T and W_a are learnable weight matrices and d_k is the dimension of the key vectors. This scaling is done to improve numerical stability as the dimension of keys, values, and queries grows [Vaswani et al., 2017].

2.5 CNNs as Alternatives to RNNs

Another extremely popular neural network architecture is the convolutional neural network (CNN). This is a variant of the FFNN that uses convolution operations to extract patterns from data that is nearby. CNNs have originally gained popularity among machine learning problems that deal with images such as image classification [LeCun et al., 1998, Krizhevsky et al., 2012, Simonyan and Zisserman, 2014]. These networks use convolution operations that are good at extracting patterns in the patches of the data they're applied on.

The key point of interest for us is that recently there has been a trend in applying CNNs for NLP tasks in order to create architectures that overcome some of the downsides of the standard RNN based architectures. The motivation for this trend is very similar to the motivation for the creation of Self-Attention Networks that are the focus of this thesis. CNN architectures like ByteNet [Kalchbrenner et al., 2016] and WaveNet [van den Oord et al., 2016] have been applied on sequence modeling tasks of character-level machine translation and text-to-speech respectively. Both models achieved SOTA results on the tasks they were tested on, beating previous RNN-based architectures such as Seq2Seq+Attention models; all while training much faster.

Figure 2.6: Figure 1 from [Kalchbrenner et al., 2016]. The architecture of ByteNet. A demonstration of the process of translating from source (in red) to target language (in blue). 1-D dilated convolutions are used for source sentence encoding.



Chapter 3

Self-Attention Networks

The focus of this thesis is a new type of neural network architecture, called self-attention networks, that use the attention mechanism discussed in Chapter 2 as the basic building block. These networks don't have any recurrence like RNNs nor utilize convolutions like CNNs. Instead, these networks apply the attention mechanism on the source input sequence directly, an operation that is called self-attention. Parts of this chapter are taken from our work for [Ambartsoumian and Popowich, 2018].

The first fully-attentional architecture, called Transformer, was recently introduced by [Vaswani et al., 2017], which utilized only the self-attention mechanism and feed-forward layers. The Transformer model achieved state-of-the-art performance on multiple machine translation datasets, without having recurrence or convolution components. Since then, self-attention networks have been successfully applied to a variety of tasks, including: image classification [Parmar et al., 2018], generative adversarial networks [Zhang et al., 2018], automatic speech recognition [Povey et al., 2017], text summarization [Liu et al., 2018], semantic role labeling [Strubell et al., 2018], as well as natural language inference and sentiment analysis [Shen et al., 2018a,b, Shen et al., 2018].

3.1 Self-Attention Mechanism

Self-attention [Lin et al., 2017], also known an intra-attention [Cheng et al., 2016b], is the process of applying the attention mechanism discussed in Section 2.4 directly on the source sequence to produce an output sequence of the same length. The attention mechanism is applied once for every sequence position by creating the query, value, and key representations from the source sequence. As a result, an input sequence $X = (x_1, x_2, ..., x_n)$ is transformed to a sequence $Y = (y_1, y_2, ..., y_n)$ where y_i incorporates the information of x_i as well as how x_i relates to all other positions in X. Formally:

$$y_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V) \tag{3.1}$$

$$\alpha_{ij} = \frac{e^{e_ij}}{\sum_{k=1}^{n} e^{e_ik}} \tag{3.2}$$

$$e_ij = score(x_iW^Q, x_jW^K) \tag{3.3}$$

$$e_ij = \frac{(x_iW^Q)(x_jW^K)^T}{\sqrt{d_y}} \tag{3.4}$$

where $x_i \in \mathbb{R}^{d_x}$, $y_i \in \mathbb{R}^{d_y}$, W^v, W^q, W^k are learnable parameters, and $score(\cdot)$ is a compatibility/similarity function such as those discussed in Section 2.4. Equation 3.7 shows the computation when scaled dot-product is chosen for the compatibility function. The query, key, and value representations for the attention calculations are created by the x_iW^q, x_jW^k, x_jW^v operations.

If the scaled dot-product (or regular dot-product) similarity function is used, the attention computations for the entire source sequence can be done in parallel by grouping the queries, keys, and values in Q, K, V matrices:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{3.5}$$

This allows the similarity scores to be computed with just one big matrix multiply, which is much faster than additive attention as that requires a feed-forward layer to be applied for every query-key pair.

3.2 Transformer Architecture

The research in this thesis was primarily inspired by this architecture. The Transformer architecture achieved impressive SOTA results for multiple machine translation datasets [Vaswani et al., 2017].

3.2.1 Motivation

The main motivation for the Transformer model was the compelling properties of the attention mechanism over the properties of recurrence and convolution mechanisms. They have looked at three properties for the recurrence, convolution, and self-attention layers. The first property is the computation complexity for every layer, the second is how much computation can be parallelized by a measure of the minimum number of sequential operations required to encoder/decode a sequence, and the third is the maximum length path between long-range dependencies of any two sequence positions. The last metric is particularly important as it demonstrates the amount of sequential computation steps that are required for a sequence encoder to create a hidden vector representation of the entire sequence, hence processing the two sequence positions that are the furthest apart. As discussed in Chapter 2, generally the more computation steps a network has to do, the harder for it to train due to increased number of calculations for the forward pass as well as for back-propagation [Hochreiter, 2001]. Table 3.1 shows these complexity metrics for each layer type. In Table 3.1, n is the sequence length (also the maximum path length as the first and last positions are the furthest away), d is the dimension size for each sequence step, and k is the kernel size of convolutions.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$

Table 3.1: from [Vaswani et al., 2017]: "Shows the maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, and k is the kernel size of convolutions."

The main takeaway is that self-attention take constant number, *O(1)*, of sequential operations to connect the first and last position of a sequence. As discussed in Chapter 2, a recurrent layer requires *O(n)* to encode a sequence and standard CNNs require a stack of *O(n/k)* convolutional layers with kernel width of size k.

3.2.2 Architecture

Transformer is an encoder-decoder model that was designed for NMT, a sequence to sequence problem. Just like previous NMT encoder-decoder systems, the encoder processes the source language sentence, and then the decoder uses the output of the encoder to construct a sentence in the target language. Unlike RNNs, the encoder doesn't produce a fixed length vector that represents the input sequence. Instead, the encoder outputs a sequence of the same length as the input, except one that is "transformed" by the encoder self-attention layers.

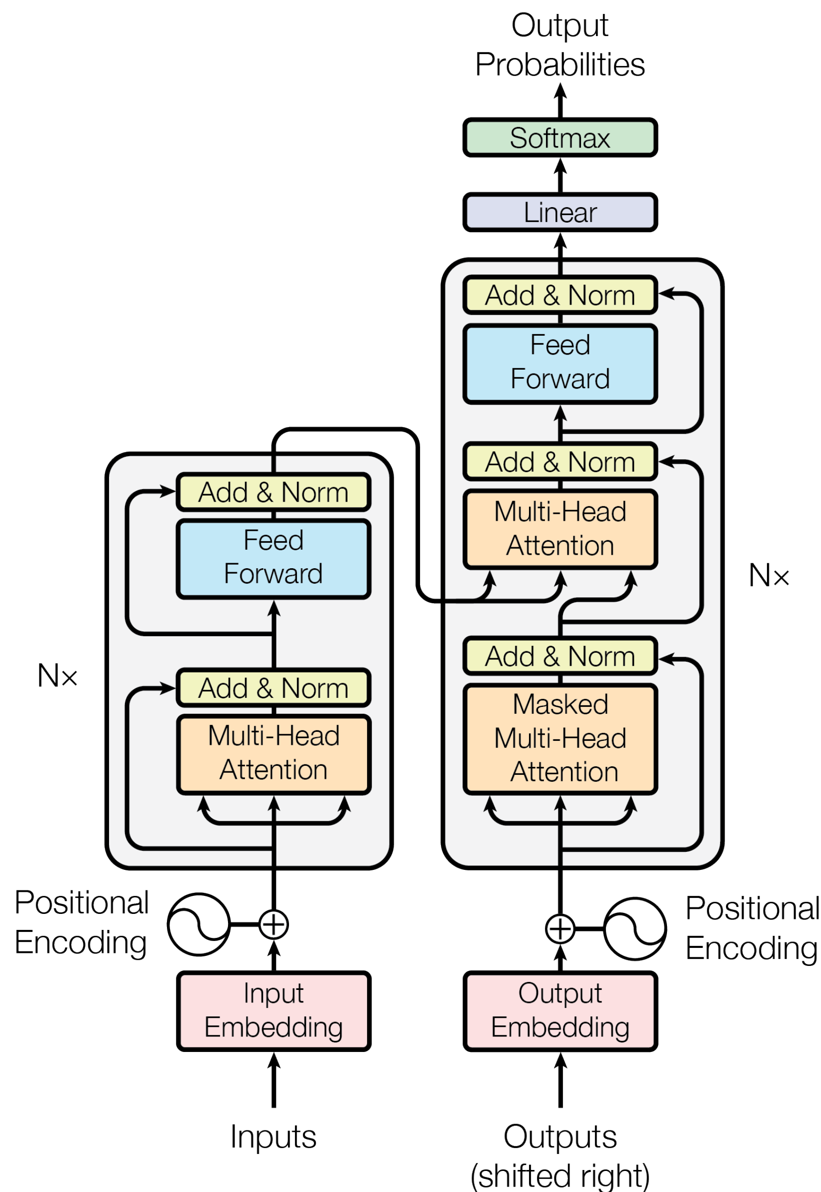
First, Transformer intakes input embeddings, such as pre-trained word embeddings, and adds positional information to them in a form of fixed absolute positional encodings. This is an important step as the self-attention mechanism doesn't inherently model the ordering of the input sequence (unlike RNNs and CNNs). We discuss positional encodings as well as other techniques of making self-attention networks model positional information in Section 3.3.

Next, the input sequence is passed through N identical encoder layers, with each having two sublayers of multi-head self-attention and feed-forward layers. Multi-head attention is discussed in the next section. Transformer utilizes some of the new deep learning techniques that allow the training of deeper models. Techniques such as residual connections [He et al., 2016] and layer normalization [Ba et al., 2016]. These techniques have been developed to help deep models train better. The residual connections also allow the positional encoding signal to propagate deeper through the network. Each sub-layer is wrapped by these operations. Thus the output of each sublayer is $LayerNorm(x + SubLayer(x))$, where $SubLayer(x)$ is either multi-head or feed-forward layer.

Just as for RNN-based encoder-decoder architectures, the decoder is still auto-regressive; it uses the previously generated words as input to generate the next word. Computationally, the decoder layers are identical to the encoder layers except that it has a second multi-head attention layer that attends over the encoder output. An important implementation detail about the decoder is that the

first multi-head attention layer needs to be masked to not allow information to attend to subsequent positions. For example, when decoding output sequence y_k, the input embeddings sequence for the decoder should be $[y_1, y_2, ..., y_{k-1}, 0's, 0's, ..., 0's]$ where $0's$ are vectors of zeros with same dimensionality as y_i.

Figure 3.1: Figure 1 from [Vaswani et al., 2017]. Visualization of the Transformer model architecture computation graph.



Multi-Head Attention

One of the key contributions of [Vaswani et al., 2017] is the introduction of the multi-head attention. Instead of performing self-attention once for (Q,K,V) of dimension d_{model}, multi-head performs attention h times on projected (Q,K,V) matrices of dimension d_{model}/h. For each head, the (Q,K,V)

matrices are uniquely projected to dimension d_{model}/h and self-attention is performed to yield an output of dimension d_{model}/h. The outputs of each head are then concatenated, and once again a linear projection layer is applied, resulting in an output of same dimensionality as performing self-attention once on the original (Q,K,V) matrices. This process is described by the following formulas:

$$\begin{aligned} \text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$. The idea here is that different heads will learn to pay attention to different features of the sequence. For example, [Vaswani et al., 2017] found that one of the attention heads in the Transformer encoder appeared to have been performing anaphora resolution, the task of determining what pronouns or nouns refer to.

3.3 Position Information Techniques

The attention mechanism is completely invariant to sequence ordering, thus self-attention networks need a method to incorporate positional information. Three main techniques have been proposed to solve this problem: adding sinusoidal positional encodings or learned positional encoding to input embeddings, or using relative positional representations in the self-attention mechanism.

Sinusoidal Position Encoding

This method was proposed by [Vaswani et al., 2017] to be used for the Transformer model. Here, positional encoding (PE) vectors are created using sine and cosine functions of different frequencies and then are added to the input embeddings. Thus, the input embeddings and positional encodings must have the same dimensionality of d_{model}. The following sine and cosine functions are used:

$$\begin{aligned} PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{\text{model}}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{\text{model}}}) \end{aligned}$$

where pos is the sentence position and i is the dimension. Using this approach, sentences longer than those seen during training can still have positional information added. We will be referring to this method as PE. The authors of this technique liked the simplicity of this method, as there are no learnable parameters.

Learned Position Encoding

In a similar method, learned vectors of the same dimension size, that are also unique to each position can be added to the input embeddings instead of sinusoidal positional encodings [Gehring et al., 2017]. There are two downsides to this approach. First, this method cannot handle sentences that are longer than the ones in the training set as no vectors are trained for those positions. Second, the further position will likely not get trained as well if the training dataset has more short sentences than longer ones. Vaswani et al. [2017] also reported that these perform identically to the positional encoding approach.

Relative Position Representations

Relative Position Representations (RPR) was introduced by [Shaw et al., 2018] as a replacement of positional encodings for the Transformer. Using this approach, the Transformer was able to perform even better for NMT, further advancing SOTA for multiple datasets, while increasing training time by only 7%. Out of the three discussed, we have found this approach to work best and we will be referring to this method as RPR throughout this thesis.

For this method, the self-attention mechanism is modified to explicitly learn the relative positional information between every two sequence positions. As a result, the input sequence is modeled as a labeled, directed, fully-connected graph, where the edges of the graph represent relative distance between sequence positions. A tunable parameter k is also introduced that limits the maximum distance considered between two sequence positions. [Shaw et al., 2018] hypothesized that this will allow the model to generalize to longer sequences at test time.

The distance between vectors x_i and x_j is modeled by vectors $a_{ij}^V, a_{ij}^K \in \mathbb{R}^{d_y}$. Equations 3.1 and 3.7, the weighted sum operation and scaled dot-product compatibility function, are modified like so:

$$y_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V + a_{ij}^V) \tag{3.6}$$

$$e_ij = \frac{(x_i W^Q)(x_j W^K + a_{ij}^K)^T}{\sqrt{d_y}} \tag{3.7}$$

Intuitively, the weighted sum and compatibility function operations can now consider the modeled distance between sequence positions. Furthermore, [Shaw et al., 2018] hypothesized that for many tasks relative position information beyond a certain distance is not useful. Thus, the maximum relative position considered is clipped to an absolute value k. Each $a_{ij}^V)$ and a_{ij}^K now maps to:

$$a_{ij}^K = w_{clip(j-i,k)}^K \tag{3.8}$$

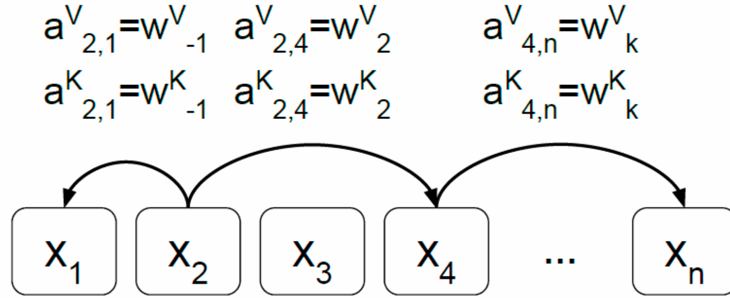
$$a_{ij}^V = w_{clip(j-i,k)}^V \tag{3.9}$$

$$clip(x,k) = max(-k, min(k,x)) \tag{3.10}$$

The final learnable parameters are $w^K = (w^K_{-k}, ..., w^K_k)$ and $w^V = (w^V_{-k}, ..., w^V_k)$, where $w^K_i, w^V_i \in \mathbb{R}^{d_y}$.

An example of modeling relative position representations of maximum clipping distance k is provided in Figure 3.2. The example demonstrates how a^V_{ij} and a^K_{ij} edge values are clipped by mapping to vectors in w^K and w^V.

Figure 3.2: Figure 1 from [Shaw et al., 2018]. An example illustrating some of the edge values that represent relative positions between sequence positions. A clipping distance of k is chosen, where $2 <= k <= n - 4$ is assumed.



Chapter 4

Proposed Architectures & Implementation Details

This chapter will introduce two self-attention network architectures called SSAN and ESSAN. SSAN was developed for our work in [Ambartsoumian and Popowich, 2018], where the goal was to create the simplest SAN architecture possible to then be compared to basic RNN and CNN architectures. ESSAN is an extension of our SSAN architecture, where we re-introduce multi-head attention, use L2-Regularization, and replace the mechanism that creates the sequence representation from the output of the final self-attention layer. We use ESSAN in section 5.2 to achieve SOTA results on the SST-fine sentiment analysis dataset, as well as in chapter 6 for time-series prediction.

4.1 Simple Self-Attention Network (SSAN)

This architecture was designed to be as simple as possible in order to compare it to simple RNN and CNN architectures. SSAN, as seen in Figure 4.1, is basically a very stripped down version of the Transformer encoder. It contains fewer feed-forward layers than Transformer and uses no multi-head attention, residual connections, or layer normalization. The default configuration uses relative position representations (RPR) for including positional information instead of positional encodings (PE), this is because we have found PE to perform poorly on all of the, relatively small, datasets that we tested throughout this thesis. Here are the justifications/intuition for the few feed-forward layers in this architecture:

- Feed-forward layers are needed for creating Q, K, V matrices because otherwise the self-attention layer will be calculating dot-product similarity scores on the exact same vectors.
- Feed-forward after the self-attention operation prepares each transformed sequence position for the next self-attention layer, or to be used in the creation of the fixed sequence representation vector.
- Feed-forward on the averaged vectors of the last self-attention layer is responsible for creating the sequence representation.

For regularization, dropout is applied on the input embeddings EMB_i, outputs of each self-attention layer EMB'_i, and the sentence representation vector. We tune the dropout keep probability in chapters 5 and 6. We use the ReLU [Nair and Hinton, 2010] activation function as it's the most common one used for feed-forward networks. We also tried the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ activation function, which led to mostly similar performance.

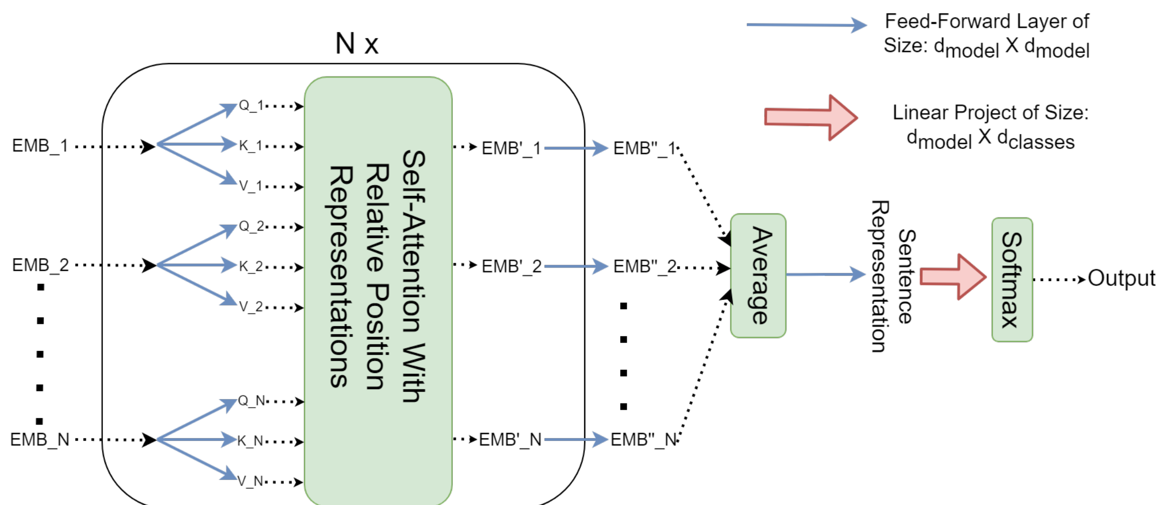


Figure 4.1: SSAN Model Architecture

4.2 Extended SSAN (ESSAN)

In order to achieve SOTA results for the SST dataset in Chapter 5, the SSAN architecture had to be slightly modified. We only needed to make the following four changes to get SOTA results for sentiment analysis in Section 5.2 and time-series prediction in Chapter 6:

- Use multi-head attention from [Vaswani et al., 2017].
- Replace the method of creating the final fixed-size sequence representation vector from the output of the last self-attention layer. We replace the averaging method from SSAN with an attention-based weighted sum approach.
- Use L2-regularization.
- Use the Swish [Ramachandran et al., 2017] activation function.

We suspected that averaging the final vector representations is not ideal as not all sequence positions should contribute equally to the final representation. We have tried multiple ways of reducing the output vectors of the last self-attention layer into a single fixed-size vector to create a representation for the input sentence/sequence. For SSAN, we did the simplest thing of simply averaging the vectors. We also tried summing them, which yielded slightly worse results. Instead

of simply averaging or summing the output of the last self-attention layer, we thought a weighted sum of these final representations would be more appropriate. To do this, we once again utilize the attention mechanism. We construct a query vector q by passing the output of the last self-attention layer through a feed-forward layer and averaging the resulting representations. We set the key and value vectors to be the output of the last self-attention layer. Attention is performed to calculate a weighted sum of the EMB'' vectors, and the result is passed through a feed-forward layer to create the final sequence representation. This process can be described with the following steps:

1. Create query vector $q = Average([FF_q(EMB''_1), FF_q(EMB''_2), ..., FF_q(EMB''_N)])$
2. Define keys and values as: $keys = values = [EMB''_1, EMB''_2, ..., EMB''_N]$
3. $SentenceRepresentation = FF_{sr}(Attention(q, keys, values))$

where FF_q and FF_{sr} are feed-forward layers. This technique was useful for the expanded SST datasets that are used in chapter 5.2.

For this model we also utilized L2-regularization, also known as weight decay, to further regularize the model and prevent over-fitting. This is a common technique in machine learning that aims to reduce the model's complexity by preventing the weights becoming too large. This is done by adding a weight penalty to the cost function:

$$C_{final}(\hat{Y}, Y) = C(\hat{Y}, Y) + \gamma \frac{1}{n} \sum_{i=1}^{n} w_i^2 \tag{4.1}$$

where $C(\hat{Y}, Y)$ is the cost function for network output $\hat{Y}$ and expected output Y, w_i is a learnable weight in the network, and γ is the L2 regularization decay factor. The final cost function, shown in equation 4.1, increases as the weights increase, thus penalizing the model for very large weights in the network. To regularize this model, the dropout keep probability as well as the L2 regularization decay factor γ need to be tuned.

For ESSAN we use the recently introduced Swish [Ramachandran et al., 2017] activation function instead of the ReLU we used for SSAN. We chose Swish because we never saw any of our network variations perform worse when switching from ReLU to Swish, while we did at times see performance decrease going from Swish to ReLU.

4.2.1 Masking

Neural network models are most commonly trained in batches to speed up training. Some complications in straightforward implementations arise when training self-attention networks. This is because these models are often training on GPUs, and training from one example at a time results in most time being spent moving data from main memory to GPU VRAM rather than performing the calculations required for training.

When dealing with datasets containing sequences of varied length, such as sentences, the training batches will most likely also contain sample sequences of varied length. The most straightforward

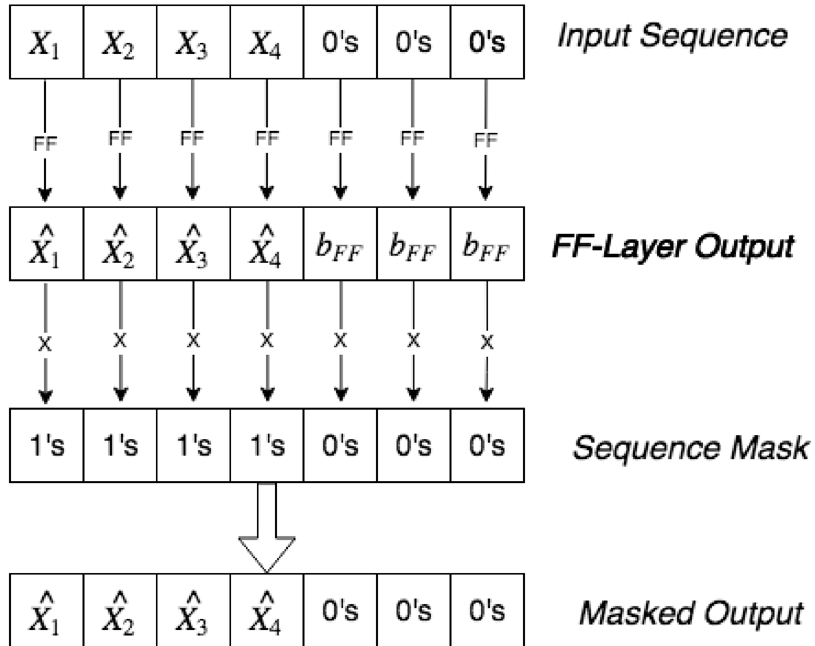


Figure 4.2: Masking sequence after feed-forward layers example.

implementation of dealing with this is to fix the *sequence_length* dimension to be the same as the longest sequence in the dataset or the longest for the batch. Then for all the shorter sequences, we simply pad them with trailing zeroes for the positions that don't exist. This becomes problematic because some of the operations in SANs will result those positions to become non-zeroes throughout the computation graph. For instance, any feed-forward layers with biases will cause such a change as demonstrated in Figure 4.2. This is problematic since this is a deviation from the theoretical model, where the learnable parameters deeper in the network are affected by these invalid positions. To fix this, we need to reset such invalid positions for each sentence after every feed-forward layer. We achieve this by calculating a mask tensor from the input tensor that determines the valid positions for each sequence. Then after every feed-forward layer, we simply do a point-wise multiplication with the mask tensor to reset the invalid positions back to the zero vector. Figure 4.2 demonstrates this entire process for a sequence $X = (X_1, X_2, X_3, X_4)$ that goes through a feed-forward layer: $\hat{X} = F(X * W_{FF} + b_{FF})$, where F is an activation function, and W_{FF}, b_{FF} are the learnable parameters.

Most other implementations we've seen do not bother with this detail, but some do mitigate the possible negative effect of this by having the *sequence_length* dimension be dynamically set per batch and possibly grouping sequences of similar length. However, most do the easiest method of fixing the maximum sequence length to be the same as the longest sequence in the dataset.

We've found that applying the sequence mask leads to slightly better performance. However, we apply this method only for ESSAN and not SSAN. This is because we did not have the space required

in our paper [Ambartsoumian and Popowich, 2018] to describe this implementation detail. Thus, to stick with our theme of keeping the SSAN description and implementation as simple as possible, we chose not to apply the sequence mask. For ESSAN, we apply the sequence mask after every feed-forward layer, which are the layers that create Q, K, V, EMB'', and $Sentence_Representation$ tensors.

Chapter 5

Sentiment Analysis

Sentiment analysis, also know as opinion mining, deals with determining the opinion classification of a piece of text. Most commonly the classification is whether the writer of a piece of text is expressing a position or negative attitude towards a product or a topic of interest [Pang and Lee, 2008]. Having more than two sentiment classes is called fine-grained sentiment analysis with the extra classes representing intensities of positive/negative sentiment (e.g. very-positive) and/or the neutral class. This field has seen much growth for the past two decades, with many applications and multiple classifiers proposed [Mäntylä et al., 2018]. Sentiment analysis has been applied in areas such as social media [Jansen et al., 2009], commerce [Jansen et al., 2009], and health care [Greaves et al., 2013b] [Greaves et al., 2013a].

In the past few years, neural network approaches have consistently advanced the state-of-the-art technologies for sentiment analysis and other natural language processing (NLP) tasks. For sentiment analysis, the neural network approaches typically use pre-trained word embeddings such as word2vec [Mikolov et al., 2013] or GloVe[Pennington et al., 2014] for input, which get processed by the model to create a sentence representation that is finally used for a softmax classification output layer. The main neural network architectures that have been applied for sentiment analysis are recurrent neural networks(RNNs) [Tai et al., 2015] and convolutional neural networks (CNNs) [Kim, 2014b], with RNNs being more popular of the two. For RNNs, typically gated cell variants such as long short-term memory (LSTM) [Hochreiter and Schmidhuber, 1997], Bi-Directional LSTM (BiLSTM) [Schuster and Paliwal, 1997], or gated recurrent unit (GRU) [Cho et al., 2014a] are used.

In this chapter, our goal is to accomplish two tasks: demonstrate that self-attention is a better building block for sequence modeling (in the context of sentiment analysis) than recurrence and convolutions, and achieve SOTA accuracy on the popular *SST* dataset using our proposed ESSAN architecture.

In section 5.1 we present the work we have done for [Ambartsoumian and Popowich, 2018]. For this section, our goal is to explore the performance and characteristics of the self-attention, recurrence, and convolution building blocks. Thus, only the most basic architectures for each category are considered. We compare our proposed SSAN architecture to LSTM, BiLSTM, and CNN architectures from [Barnes et al., 2017]. We compare the classification accuracy on six sentiment

analysis datasets as well as explore other architecture characteristics such as train and inference times, number of trainable parameters, and VRAM consumption during training. In this section we also explore the effectiveness of various techniques of incorporating positional information into SANs.

In section 5.2, we demonstrate that our proposed ESSAN architecture achieves SOTA accuracy on the SST-fine, beating the previous SOTA SAN model called DiSAN [Shen et al., 2018a]. ESSAN does this while having less trainable parameters and consuming much less VRAM during training than DiSAN.

Parts of this chapter are taken from our work for [Ambartsoumian and Popowich, 2018].

5.1 Comparison of Building Blocks: SAN vs RNN vs CNN

In this section we demonstrate that self-attention is a better building block compared to recurrence or convolutions for sentiment analysis classifiers. We extend the work of [Barnes et al., 2017] by exploring the behaviour of various self-attention architectures, such as the proposed *SSAN* from Chapter 4, on six datasets and making direct comparisons to their work. We set our baselines to be their results for *Lstm*, *BiLstm*, and *Cnn* models, and used the same code for dataset pre-processing, word embedding imports, and batch construction. Finally, we explore the effectiveness of SAN architecture variations such as different techniques of incorporating positional information into the network, using multi-head attention, and stacking self-attention layers. Our results suggest that relative position representations is superior to positional encodings, as well as highlight the efficiency of the stacking self-attention layers. Code for the work done in this section is open-sourced here [1].

5.1.1 Experiments

To reduce implementation deviations from previous work, we use the codebase from [Barnes et al., 2017] and only replace the model and training process. We re-use the code for batch pre-processing and batch construction for all datasets, accuracy evaluation, as well as use the same word embeddings[2]. All neural network models use cross-entropy for the training loss.

All experiments and benchmarks were run with the same hardware and software versions. For model implementations: LSTM, BiLSTM, and CNN baselines are implemented in Keras [Chollet et al., 2015] with Tensorflow backend. All self-attention models are implemented in Tensorflow. Details for software and hardware used are in Table 5.1.

5.1.2 Datasets Used

In order to determine if certain neural network building blocks are superior, we test on the six datasets from [Barnes et al., 2017]. The macro-average accuracy is calculated to better understand how all the

[1] https://github.com/Artaches/SSAN-self-attention-sentiment-analysis-classification

[2] https://github.com/jbarnesspain/sota_sentiment

Hardware	Details	Software	Version
CPU	Intel i7 5820k @ 3.3Ghz	Tensorflow	1.7
RAM	32GB DDR4-2133	Keras	2.0.8
GPU	Nvidia GTX 1080 w/ 8GB GDDR5 VRAM	CUDA	9.1
		CuDNN	5.1.5
		OS	Ubuntu 16.04.4 LTS

Table 5.1: Details of hardware and software that were used for all experiments.

architectures perform on datasets with different properties. The summary for dataset properties is in Table 5.2.

The Stanford Sentiment Treebank (*SST-fine*) [Socher et al., 2013] deals with movie reviews, containing five classes [very-negative, negative, neutral, positive, very-positive]. (*SST-binary*) is constructed from the same data, except the neutral class sentences are removed, all negative classes are grouped, and all positive classes are grouped. However, the *SST-fine* variant is the more popular version, with some papers often not reporting results for the binary variant [Shen et al., 2018a]. For the experiments in this section, the datasets are pre-processed to only contain sentence-level labels, and none of the models reported in this work utilize the phrase-level labels that are also provided. This is the most popular sentiment analysis dataset of the group, with many papers pushing its SOTA results over the years [Zhou et al., 2016, Wang et al., 2016, Kokkinos and Potamianos, 2017, Shen et al., 2018a].

The *OpeNER* dataset [Agerri et al., 2013] is a dataset of hotel reviews with four sentiment classes: very negative, negative, positive, and very positive. This is the smallest dataset with the lowest average sentence length.

The SenTube datasets [Uryupina et al., 2014] consist of YouTube comments with two sentiment classes: positive and negative. The SenTube-A dataset contains comments that relate to automabiles, while the SenTube-T contains comments relating to tablets. The datasets were constructed by manually extracting comments with sentiment polarity towards a product, taken from commercial videos for the product types. These datasets contain the longest average sentence length as well as the longest maximum sentence length of all the datasets.

The SemEval Twitter dataset (*SemEval*) [Nakov et al., 2013] consists of tweets with three classes: positive, negative, and neutral. This dataset was created for the SemEval 2013 shared task B competition. An interesting fact for this dataset is that the SOTA result for the competition was achieved by an SVM [Hearst, 1998] model, with an accuracy of around 69.5%. During our testing for the results in Table 5.3, we often saw best models that achieved an accuracy of over 73%, with many achieving 70% as the average accuracy. In our opinion, this demonstrates why neural network approaches that utilize pre-trained word embeddings have become so popular in the past few years for many NLP tasks.

	Train	Dev.	Test	# of Classes	Average Sent. Length	Max Sent. Length	Vocab. Size	Wiki Emb. Coverage	300D Emb. Coverage
SST-fine	8,544	1,101	2,210	5	19.53	57	19,500	94.4%	89.0%
SST-binary	6,920	872	1,821	2	19.67	57	17,539	95.0%	89.6%
OpeNER	2,780	186	743	4	4.28	23	2,447	94.2%	99.3%
SenTube-A	3,381	225	903	2	28.54	127	18,569	75.6%	74.5%
SenTube-T	4,997	333	1,334	2	28.73	121	20,276	70.4%	76.0%
SemEval	6,021	890	2,376	3	22.40	40	21,163	77.1%	99.8%

Table 5.2: Modified Table 2 from [Barnes et al., 2017]. Dataset statistics, embedding coverage of dataset vocabularies, as well as splits for Train, Dev (Development), and Test sets. The 'Wiki' embeddings are the 50, 100, 200, and 600 dimension used for experiments.

5.1.3 Pre-Trained Word Embeddings

We use the exact same word embeddings as [Barnes et al., 2017]. They trained the 50, 100, 200, and 600-dimensional word embeddings using the word2vec algorithm described in [Mikolov et al., 2013] on a 2016 Wikipedia dump. In order to compare to previous work, they also used the publicly available Google 300-dimensional word2vec embeddings [3], which are trained on a part of Google News dataset. For all models, out-of-vocabulary words are initialized randomly from the uniform distribution on the interval [-0.25 , 0.25]. Vocabulary coverage for each dataset & word embedding pair can be seen in Table 5.2. The 300D google embeddings have better coverage than the Wikipedia based ones on all datasets except SST-fine and SST-binary.

5.1.4 Baselines Descriptions

We take six classifiers from [Barnes et al., 2017] and use their published results as baselines. Two of the methods are based on logistic regression, *Bow* and *Ave*, and 3 are neural network based, *Lstm*, *BiLstm*, and *Cnn*.

The (*Bow*) baseline is a L2-regularized logistic regression trained on bag-of-words representation. Each word is represented by a one-hot vector of size $n = |V|$, where $|V|$ is the vocabulary size.

The (*Ave*) baseline is also a L2-regularized logistic regression classifier except trained on the average of the 300-dimension word embeddings for each sentence.

The *Lstm* baseline, input word embeddings are passed into an LSTM layer. Then a 50-dimensional feed-forward layer with ReLU activations is applied, followed by a softmax layer that produces the classification output. Dropout [Srivastava et al., 2014] is applied to the input word embeddings for regularization.

The *BiLstm* baseline is the same as *Lstm*, except that a second LSTM layer is used to process the input word embeddings in the reverse order. The outputs of the two LSTM layers are concatenated

[3] `https://code.google.com/archive/p/word2vec/`

and passed a feed-forward layer, following by the output softmax layer. Dropout is applied identically as in *Lstm*. This modification improves the networks ability to capture long-range dependencies.

The final baseline is a simple *Cnn* network. The input sequence of n embeddings is reshaped to an $n \times R$ dimensional matrix M, where R is the dimensionality of the embeddings. Convolutions with filter size of [2,3,4] are applied to M, following by a pooling layer of length 2. As for *Lstm* networks, a feed-forward layer is applied followed by an output softmax layer. Here, dropout is applied to input embeddings as well as after the convolution layers.

The *Lstm*, *BiLstm*, and *Cnn* baselines are trained using ADAM [Kingma and Ba, 2014] with cross-entropy loss and mini-batches of size 32. Hidden layer dimension, dropout amount, and the number of training epochs are tuned on the validation set for each (model, input embedding, dataset) combination.

5.1.5 Self-Attention Architectures

We use *1-Layer SSAN + RPR* and *2-Layer SSAN + RPR* to compare the self-attention mechanism to the recurrence and convolution mechanisms in *LSTM*, *BiLSTM*, and *CNN* models. We compare these models using all word embeddings sizes.

Next, we explore the performance of a modified *Transformer Encoder* described in Chapter 4. We do this to determine if a more complex architecture that utilized multi-head attention is beneficial.

Finally, we compare the performance of using positional encodings (*+PE*) and relative positional representations (*+RPR*) for the *Transformer Encoder* and *1-Layer-SSAN* architectures. We also test *1-Layer SSAN* without using any positional information techniques.

For the self-attention networks, we simplify the training process to tune only one parameter and apply the same process to all models. Only the learning rate is tuned for every (model, input embedding) pair. We fix the number of batches to train for to 100,000 and pick the model with highest validation accuracy. Each batch is constructed by randomly sampling the training set. Model dimensionality d_{model} is fixed to being the same as the input word embeddings. Learning rate is tuned based on the size of d_{model}. For d_{model} dimensions [50, 100, 200, 300, 600] we use learning rates of [0.15, 0.125, 0.1, 0.1, 0.05] respectively, because the larger d_{model} models tend to over-fit faster. Dropout of 0.7 is applied to all models, and the ADADELTA [Zeiler, 2012] optimizer is used with cross-entropy loss.

5.1.6 Analysis

Table 5.3 contains the summary of all the experimental results. For all neural network models we report mean test accuracy of five runs as well as the standard deviations. Macro-Avg results are the average accuracy of a model across all datasets. We focus our discussion on the Macro-Avg column as it demonstrates the models general performance for sentiment analysis.

Our results show general better performance for self-attention networks in comparison to *Lstm*, *BiLstm* and *Cnn* models. Using the same word embedding, all of the self-attention models receive

higher Macro-Avg accuracy than all baseline models. *1-Layer-SSAN+RPR* models generally perform the best for all (input embeddings, dataset) combinations, and getting top scores for five out of six datasets. *Transformer Encoder+RPR* also performs comparatively well across all datasets, and achieves top accuracy for the *OpeNER* dataset.

Using *2-Layer-SSAN+RPR* does not yield better performance results compared to *1-Layer-SSAN+RPR*. We believe that one self-attention layer is sufficient as the datasets that we have tested on were relatively small. This is reinforced by the results we see from *Transformer Encoder + RPR* since it achieves similar accuracy as *2-Layer-SSAN+RPR* and *1-Layer-SSAN+RPR* while having greater architectural complexity and more trainable parameters, see Table 5.4.

Using relative positional representations for *1-Layer-SSAN+RPR* increases the Macro-Avg accuracy by 2.8% compared to using positional encodings for *1-Layer-SSAN+PE*, and by 0.9% compared to using no positional information at all (*1-Layer-SSAN*). Interestingly enough, we observe that using no positional information performs better than using positional encodings. This could be attributed once again to small dataset size, as [Vaswani et al., 2017] successfully used positional encodings for larger MT datasets.

Another observation is that SenTube dataset trials achieve a low accuracy despite having binary classes. This is unexpected as generally with a low number of classes it is easier to train on the dataset and achieve higher accuracy. We suspect that this is because SenTube contains longer sentences and very low word embedding coverage. Despite this, SSANs perform relatively well on the SenTube-A dataset, which suggests that they are superior at capturing long-range dependencies compared to other models.

Smaller d_{model} *SSAN* models perform worse for lower dimension input embeddings on *SST-fine*, *SST-binary* and *OpeNER* datasets while still performing well on *SenTube* and *SemEval*. This is caused by the limitations of our training process where we forced the network d_{model} to be same as the input word embeddings and use the same learning rate for all datasets. We found that working with smaller dimensions of d_{model} the learning rate needed to be tuned individually for some datasets. For example, using a learning of 0.15 for 50D models would work well for *SenTube* and *SemEval*, but would under-fit for *SST-fine*, *SST-binary* and *OpeNER* datasets. We decided to not modify the training process for the smaller input embeddings in order to keep our training process simplified.

5.1.7 Model Characteristics

Here we compare training and test efficiency, memory consumption and number of trainable parameters for every model. For all models, we use the *SST-fine* dataset, hidden dimension size of 300, Google 300D embeddings, batch sizes of 32 for both training and inference, and the ADAM optimizer [Kingma and Ba, 2014]. The *Training Time* test is the average time it takes every model to train on 10 epochs of the SST-fine train set (2670 batches of size 32). The *Inference Time* test is the average time it takes a model to produce predictions for the validation set 10 times (344 batches of size 32). Table 5.4 contains the summary of model characteristics. The GPU VRAM usage is the amount of GPU video memory that is used during training.

	Model	Dim.	SST-fine	SST-binary	OpeNER	SenTube-A	SenTube-T	SemEval	Macro-Avg.
Baselines	*Bow*		40.3	80.7	77.1	60.6	66.0	65.5	65.0
	Ave	300	41.6	80.3	76.3	61.5	64.3	63.6	64.6
	Lstm	50	43.3 (1.0)	80.5 (0.4)	81.1 (0.4)	58.9 (0.8)	63.4 (3.1)	63.9 (1.7)	65.2 (1.2)
		100	44.1 (0.8)	79.5 (0.6)	82.4 (0.5)	58.9 (1.1)	63.1 (0.4)	67.3 (1.1)	65.9 (0.7)
		200	44.1 (1.6)	80.9 (0.6)	82.0 (0.6)	58.6 (0.6)	65.2 (1.6)	66.8 (1.3)	66.3 (1.1)
		300	45.3 (1.9)	81.7 (0.7)	82.3 (0.6)	57.4 (1.3)	63.6 (0.7)	67.6 (0.6)	66.3 (1.0)
		600	44.5 (1.4)	83.1 (0.9)	81.2 (0.8)	57.4 (1.1)	65.7 (1.2)	67.5 (0.7)	66.5 (1.0)
	BiLstm	50	43.6 (1.2)	82.9 (0.7)	79.2 (0.8)	59.5 (1.1)	65.6 (1.2)	64.3 (1.2)	65.9 (1.0)
		100	43.8 (1.1)	79.8 (1.0)	82.4 (0.6)	58.6 (0.8)	66.4 (1.4)	65.2 (0.6)	66.0 (0.9)
		200	44.0 (0.9)	80.1 (0.6)	81.7 (0.5)	58.9 (0.3)	63.3 (1.0)	66.4 (0.3)	65.7 (0.6)
		300	45.6 (1.6)	82.6 (0.7)	82.5 (0.6)	59.3 (1.0)	66.2 (1.5)	65.1 (0.9)	66.9 (1.1)
		600	43.2 (1.1)	83.0 (0.4)	81.5 (0.5)	59.2 (1.6)	66.4 (1.1)	68.5 (0.7)	66.9 (0.9)
	CNN	50	39.9 (0.7)	81.7 (0.3)	80.0 (0.9)	55.2 (0.7)	57.4 (3.1)	65.7 (1.0)	63.3 (1.1)
		100	40.1 (1.0)	81.6 (0.5)	79.5 (0.9)	56.0 (2.2)	61.5 (1.1)	64.2 (0.8)	63.8 (1.1)
		200	39.1 (1.1)	80.7 (0.4)	79.8 (0.7)	56.3 (1.8)	64.1 (1.1)	65.3 (0.8)	64.2 (1.0)
		300	39.8 (0.7)	81.3 (1.1)	80.3 (0.9)	57.3 (0.5)	62.1 (1.0)	63.5 (1.3)	64.0 (0.9)
		600	40.7 (2.6)	82.7 (1.2)	79.2 (1.4)	56.6 (0.6)	61.3 (2.0)	65.9 (1.8)	64.4 (1.5)
Self-Attention Models	*1-Layer SSAN + RPR*	50	42.8 (0.8)	79.6 (0.3)	78.6 (0.5)	**64.1** (0.4)	67.0 (1.0)	67.1 (0.5)	66.5 (0.6)
		100	44.6 (0.3)	82.3 (0.3)	81.6 (0.5)	61.6 (1.3)	68.6 (0.6)	68.6 (0.5)	67.9 (0.6)
		200	45.4 (0.4)	83.1 (0.5)	82.3 (0.4)	62.2 (0.6)	68.4 (0.8)	70.5 (0.4)	68.6 (0.5)
		300	**48.1** (0.4)	**84.2** (0.4)	83.8 (0.2)	62.5 (0.3)	68.4 (0.8)	**72.2** (0.8)	**69.9** (0.5)
		600	47.7 (0.7)	83.6 (0.4)	83.1 (0.4)	62.0 (0.4)	**68.8** (0.7)	70.5 (0.8)	69.2 (0.5)
	2-Layer SSAN + RPR	50	43.2 (0.9)	79.8 (0.2)	79.2 (0.6)	63.0 (1.3)	66.6 (0.5)	67.5 (0.7)	66.5 (0.7)
		100	45.0 (0.4)	81.6 (0.9)	81.1 (0.4)	63.3 (0.7)	67.7 (0.5)	68.7 (0.4)	67.9 (0.5)
		200	46.5 (0.7)	82.8 (0.5)	82.3 (0.6)	61.9 (1.2)	68.0 (0.8)	69.6 (0.8)	68.5 (0.8)
		300	**48.1** (0.8)	83.8 (0.9)	83.3 (0.9)	62.1 (0.8)	67.8 (1.0)	70.7 (0.5)	69.3 (0.8)
		600	47.6 (0.5)	83.7 (0.4)	82.9 (0.5)	60.7 (1.4)	68.2 (0.7)	70.3 (0.3)	68.9 (0.6)
	Transformer Encoder + RPR	300	47.3 (0.4)	83.8 (0.4)	**84.2** (0.5)	62.0 (1.4)	68.2 (N1.6)	72.0 (0.5)	69.6 (0.8)
	Transformer Encoder + PE	300	45.0 (0.7)	82.0 (0.6)	83.3 (0.7)	62.3 (2.4)	66.9 (0.8)	68.4 (0.8)	68.0 (1.0)
	1-Layer SSAN	300	47.2 (0.5)	83.9 (0.7)	83.6 (0.6)	62.1 (2.5)	68.7 (1.0)	70.2 (1.2)	69.3 (1.1)
	1-Layer SSAN + PE	300	45.0 (0.3)	82.9 (0.2)	80.7 (0.6)	62.6 (2.3)	67.8 (0.4)	69.1 (0.3)	68.0 (0.7)

Table 5.3: Modified Table 3 from [Barnes et al., 2017]. The baseline results are taken from [Barnes et al., 2017]; the self-attention models results are ours. Test accuracy averages and standard deviations (in brackets) of 5 runs. Best model for each dataset is given in **bold**.

Cnn has the lowest number of parameters but consumes the most GPU memory. It also has the shortest training and inference time, which we attributed to the low number of parameters.

Using relative position representations compared to positional encoding for *1-Layer-SSAN* increases the number of trainable parameters by only 2.7%, training time by 11.2%, and inference time by 4.7%. These findings are similar to what [Shaw et al., 2018] reported.

BiLSTM has double the number of parameters as well as near double training and inference times compared to *LSTM*. This is reasonable due to the nature of the architecture being two *LSTM*

Model	# of Parameters	GPU VRAM Usage (MB)	Training Time (s)	Inference Time (s)
LSTM	722,705	419MB	235.9s	7.6s
BiLSTM	1,445,405	547MB	416.0s	12.7s
CNN	83,714	986MB	21.1s	0.85s
1-Layer SSAN + RPR	465,600	381MB	64.6s	8.9s
1-Layer SSAN + PE	453,000	381MB	58.1s	8.5s
2-Layer SSAN + RPR	839,400	509MB	70.3s	9.3s
Transformer + RPR	1,177,920	510MB	78.2s	9.7s

Table 5.4: Neural networks architecture characteristics. A comparison of number of learnable parameters, GPU VRAM usage (in megabytes) during training, as well as training and inference times (in seconds).

layers. Much like *BiLSTM*, going from *1-Layer-SSAN* to *2-Layer-SSAN* doubles the number of trainable parameters. However, the training and inference times only increase by 20.1% and 9.4% respectively. This demonstrates the efficiency of the self-attention mechanism due to it utilizing only matrix multiply operations, for which GPUs are highly-optimized.

We also observe that self-attention models are faster to train than *LSTM* by about 3.4 times, and 5.9 times for *BiLSTM*. However, inference times are slower than *LSTM* by 15.5% and faster than *BiLSTM* by 41%.

5.1.8 Discussion

In this section we focused on demonstrating that simple self-attention networks achieve better accuracy on six datasets than simple RNN and CNN architectures, and in the process agruing that self-attention is a superior mechanism. In our experiments, multiple *SSAN* networks performed better than *Cnn* and *Lstm* models; Self-attention architecture resulted in higher accuracy than LSTMs while having 35% fewer parameters and shorter training time by a factor of 3.5. Additionally, we showed that SSANs achieved higher accuracy on the *SenTube* datasets, which suggests they are also better at capturing long-term dependencies than *RNNs* and *CNNs*. Finally, we reported that using relative positional representation is superior to both using positional encodings, as well as not incorporating any positional information at all. Using relative positional representations for self-attention architectures resulted in higher accuracy with negligible impact on model training and inference efficiency.

5.2 SOTA Approach for SST Datasets

In this section we use our proposed ESSAN architecture from Chapter 4 to achieve SOTA accuracy on the *SST-fine* dataset, beating the previous best SAN architecture called Directional Self-Attention Network (DiSAN) [Shen et al., 2018a]. We use training methodology that is as close to DiSAN's as possible, using the same dataset pre-processing technique, same word embeddings, and the same

optimizer. We do this to make sure that ESSAN performs better due to architectural differences and not the training methodology. We also perform an ablation study for the methods of creating the final sequence representation from the output of the last self-attention layer. We demonstrate that the attention-based approach that was proposed for ESSAN yields improvements over the averaging method. Finally, just like we did for the previous section, we compare the performance and model characteristics of ESSAN. Compared to DiSAN, ESSAN has half the parameters, trains twice as fast, and uses four times less GPU memory during training.

5.2.1 SST Dataset Processing

Here we utilize all of the labels provided in SST, not just the sentence-level labels. We follow the same methodology the DiSAN implementation [4], that is by simply treating every labeled sub-phrase as a unique example, including sub-phrases of length 1. By doing this for the SST-fine dataset, the dataset size jumps to 318,582 examples. The class distrubution of this dataset is as follows: {*class_1=8,245, class_2=34,362, class_3=219,788, class_4=44,194, class_5=11,993*}, which is 318,582 training examples in total. There are now disproportionally more examples for $class_3$ because most sub-phrases are just single words and most of those are of neutral class. To balance out the class distribution of this new dataset, we simply add the examples for other classes multiple times by the following ratios $[6, 4, 1, 4, 6]$. This results in a dataset class distribution of: {*class_1=44,010, class_2=130,794, class_3=219,788, class_4=169,810, class_5=65,518*}, which is 629,920 total examples. Finally, to increase the number of longer sentences in the dataset, we also add the original sentence-level examples 4 extra times, resulting in the final training dataset size of 672,640.

For SST-binary dataset we apply the same dataset pre-processing method. We then add examples from the classes by multiples of $[3, 2, 1, 2, 3]$, and the original sentence-level examples 2 extra times. Then, the neutral class is dropped, and classes 1 and 2 as well as 4 and 5 are combined to create only two classes, representing positive/negative sentiment. The final class distribution is: {*class_1=92,367, class_2=123,079*}, which is 215,446 training examples in total.

5.2.2 Experiments

We compare the results of our ESSAN architecture to DiSAN, the previous reported SOTA self-attention architecture on the SST-fine dataset [Shen et al., 2018a]. Just like Transformer and our proposed architecture, DiSAN does not use any RNN and CNN components, and instead utilizes only self-attention and feed-forward layers. The key points for this architecture are: when computing attention for position x_i it uses directional masks that distinguish which sequence positions came before and after position x_i, contains a multi-dimensional self-attention layer that performs feature-wise attention. DiSAN achieved several SOTA results for SNLI [Bowman et al., 2015] and various text classification datasets, including the SST-fine.

[4] https://github.com/taoshen58/DiSAN

For ESSAN, we mimic the training procedure of DiSAN as close as possible. Just as DiSAN, AdaDelta(rho=0.95) is used for the optimizer. Word embeddings have a big effect on a model's performance, especially context-sensitive word embedding [Tai et al., 2015]. For this reason we make sure to use the same word embeddings as DiSAN, the publicly available GloVe word embeddings [5]. The out-of-vocabulary words are initialized randomly from the uniform distribution on the interval [-0.05 , 0.05]. Glorot initialization [Glorot and Bengio, 2010] is used for all weight matrices, and biases are initialized to zeros.

For ESSAN we use the following hyper-parameters for training on SST-fine and SST-binary datasets:

Dataset	**Layers**	**Heads**	**Learning Rate**	**Dropout**	**L2-Regularization**
SST-fine	2	15	1	0.5	0.0075
SST-binary	2	20	2.5	0.4	0.0025

Table 5.5: Hyper-parameters used by ESSAN for SST-fine and SST-binary datasets.

Other ESSAN hyper-parameters that are used for both datasets are: RPR clipping of length 20, Swish activation function [Ramachandran et al., 2017], d_{model} dimension is 300, batch size is 128, and total training steps = 100,000. The model with the best validation accuracy is chosen for each run.

We also perform an ablation study for the attention-based method of creating the final sequence representation vector. The *ESSAN w/o AttnSeqRep* is identical to ESSAN (same hyper-parameters) except that it uses the averaging method from SSAN to create the final sequence representation vector.

5.2.3 Results & Discussion

There is no standard way for reporting accuracy among previous works; most authors prefer to only report the best result that they were able to achieve, but some do report mean and standard deviation of usually 5 runs. Unfortunately this has led to some misleading citations among the publications. For example, DiSAN [Shen et al., 2018a] reported only the top scores, with Tree-LSTM [Tai et al., 2015] being stated to achieve 51.0%. However, in the original paper, 51% is actually the mean accuracy reported; the top score was most likely higher than 51.5% (since the reported standard deviation is 0.5%).

For ESSAN, we report the best achieved accuracy as well as the mean and standard deviation of 5 runs in Table 5.6. The runs are are done sequentially on the same machine, and are not chosen

[5] `http://nlp.stanford.edu/data/glove.6B.zip`

Model	SST-fine Top	SST-fine Avg	SST-bin Top	SST-bin Avg
MV-RNN [Socher et al., 2013]	44.4	-	82.9	-
RNTN [Socher et al., 2013]	45.7	-	85.4	-
Bi-LSTM [Li et al., 2015]	-	49.9 (0.8)	-	86.7 (0.5)
DCNN [Kalchbrenner et al., 2014]	48.5	-	86.8	-
Tree-LSTM [Tai et al., 2015]	-	51.0 (0.5)	-	88.0 (0.3)
CNN-non-static [Kim, 2014a]	48	-	87.2	-
CNN-Tensor [Lei et al., 2015]	51.2	-	88.6	-
NCSL [Teng et al., 2016]	51.1	-	89.2	-
LR-Bi-LSTM [Qian et al., 2017]	50.6	-	-	-
TreeBiGRU+Attn [Kokkinos and Potamianos, 2017]	52.4	-	89.5	-
DiSAN [Shen et al., 2018a]	51.6	51.0 (0.7)	-	87.8 (0.3) [6]
Bi-BloSAN [Shen et al., 2018b]	-	50.6 (0.5)	-	87.4 (0.2)
ESSAN w/o AttnSeqRep (Ours)	51.4	50.5 (0.8)	87.9	87.0 (0.5)
ESSAN (Ours)	52.7	51.5 (0.8)	88.5	87.5 (0.6)

Table 5.6: Test accuracy of fine-grained sentiment analysis on SST datasets. The "Top" columns contain the highest accuracy that each method reported. The "Avg" columns report mean and standard deviation (in bracket) of 5 runs.

by us from a larger sample. The best achieved accuracy reported is from the 5 runs used for mean and standard deviation. We believe reporting the results using this methodology reflects the true performance of our models more accurately than reporting the best achieved accuracy from dozens of runs.

In Table 5.6, we can see that overall ESSAN performs better than DiSAN on the SST-fine dataset (2.0% higher average accuracy) but performs slightly worse on the SST-binary dataset(0.34% lower average accuracy). ESSAN's top score for SST-fine dataset is 2.53% higher than DiSANs.

The original DiSAN paper [Shen et al., 2018a] does not include any results for the SST-bin dataset; the numbers we report are from their follow-up paper [Shen et al., 2018b] that introduced Bi-BloSAN, a more efficient SAN architecture. Note that we couldn't reproduce the numbers for DiSAN and SST-bin dataset as no information for dataset pre-processing or model hyper parameters were provided in any of the papers or their open-source implementation. The high variance of ESSAN

[6]This results was taken from [Shen et al., 2018b], a follow-up paper to DiSAN by the same authors.

Model	# of Parameters	GPU VRAM Usage (MB)	Training Time (s)	Inference Time (s)
ESSAN w/o AttnSeqRep	816,660	497MB	73.7s	10.2s
ESSAN	906,060	497MB	83.9s	11.4s
DiSAN	1,806,000	2289MB	161.8s	10.3s

Table 5.7: ESSAN vs DiSAN comparison of number of learnable parameters, GPU VRAM usage (in megabytes) during training, as well as training and inference times (in seconds).

for SST-bin dataset as well as the relatively low accuracy compared to DiSAN could be caused by the different dataset pre-processing methodology.

Finally, we can see that ESSAN attention-based method for creating the sequence representation vector from the output of the last self-attention layer is beneficial. This method yields better performance than the averaging method used by *ESSAN w/o AttnSeqRep* model; 1.98% increase for STT-fine and 0.58% increase for SST-bin datasets.

5.2.4 Model Characteristics

Just as we did for the previous section, here we explore the efficiency and other properties of ESSAN and DiSAN architectures. We use the same procedure: $d_{model} = 300$, GloVe 300D embeddings, batch_size=32, ADAM optimizer. The *Training Time* test is the average time it takes every model to train on 10 epochs of the SST-fine train set (2670 batches of size 32). The *Inference Time* test is the average time it takes a model to produce predictions for the validation set 10 times (344 batches of size 32).

From Table 5.4, we can see that ESSAN has more favourable model characteristics than DiSAN. ESSAN has half as many trainable parameters, uses 4.6x less memory during training, and has 1.92x faster training time. The efficiency differences between *ESSAN* and *ESSAN w/o AttnSeqRep* is minor, at around 11% increase in trainable parameters, 14% increase in training time, and 12% increase in inference time.

The directional masks that DiSAN utilizes causes its various efficiency metrics to grow quadratically with respect to d_{model}. ESSAN on the other hand, is still essentially a simplified transformer encoder, thus has similar efficiency benefits. The authors of DiSAN did follow up with a more efficient architecture called Bi-BloSAN that has essentially the same efficiency as Transformer [Shen et al., 2018b]. However, we can see in Table 5.6 that increase in efficiency comes at a significant accuracy decrease, further expanding the accuracy gap between ESSAN for the SST-fine dataset.

Chapter 6

Time-Series Regression

In this section we apply our ESSAN architecture for a completely different problem, multivariate time-series prediction. Time-series prediction is a regression problem of trying to predict a quantity based on some input. In the case of input consisting of multiple variables, the problem is called multivariate regression. A simple example of regression is trying to predict tomorrow's price of a company's stock on some stock-market based on the closing prices for the past ten days. An example of multivariate regression would be using not just closing prices, but also opening, high, low, and quantity sold; thus 5 variables per day.

We show our preliminary results on one dataset, called SML2010, and demonstrate that our ESSAN architecture achieved SOTA results for 3 different metrics, beating the previous best RNN-based approaches. We directly compare our results to that of [Qin et al., 2017] by using their reported results for various RNN architectures as baselines. For future work, we also plan to test ESSAN on the second dataset from [Qin et al., 2017], called NASDAQ100. We omitted our preliminary results for this dataset due to time constraints, and plan to finish our implementation for it as future work.

A NARX (nonlinear autoregressive exogenous) model aims to forecast the value of a target feature at time t based on the historic values up to time $t-1$ as well as values for other features for up to time t. Formally, NARX aims to learn the mapping function that computes: $\hat{y}_t = F(y_1, .., y_{t-1}, x_1, .., x_t)$, where $y_i \in \mathbb{R}$, $x_i \in \mathbb{R}^n$, and n is the number of other driving time-series features.

6.1 Dataset Description

Dataset	Driving Series	Size		
		Train	Validation	Test
SML 2010	17	3,200	400	537

Table 6.1: The driving series information for number of driving series and train/validation/test splits for the SML 2010 dataset.

The dataset we test on is the SML 2010 dataset [Zamora-Martínez et al., 2014], where the task is forecasting indoor temperature. The data was collected from a monitor system mounted in a domestic house. There is one categorical (day of the week) and 17 driving series that measure features like: outdoor temperature, carbon dioxide levels in the rooms, room humidity, etc. The target series for this dataset is the indoor room temperature of the house. The data was sampled every minute and smoothed with 15 minute means, containing a total of 40 days of data, which is 4137 time steps. We use the Train/Validation/Test splits of 3200/400/537, same as [Qin et al., 2017]. Table 6.1 summarizes the description of this dataset.

We normalize the dataset by feature-scaling using the Min-Max method. For each feature, we calculate the minimum and maximum values from the training set. Using these values, we normalize the train/validation/test data splits to the range [0, 0.75]. This process for feature X is described by the following equations:

$$X_{min} = Minimum(Train_X) \tag{6.1}$$

$$X_{max} = Maximum(Train_X) \tag{6.2}$$

$$X' = a + \frac{(X - X_{min})(b - a)}{X_{max} - X_{min}} \tag{6.3}$$

where a=0, and b=0.75 in order to normalize in the range [0, 0.75]. For calculating the performance metrics, we then de-normalize the output of our model X'_{pred} like so:

$$X_{pred} = \frac{((X'_{pred} - a)(X_{max} - X_{min}))}{b - a} + X_{min} \tag{6.4}$$

The metrics described in section 6.3 are calculated using the de-normalized output X_{pred}.

For DA-RNN, [Qin et al., 2017] normalizes each feature via standardization using the mean and variance. For ESSAN, the standardization method at times caused worse performance as well as made ESSAN more sensitive to hyper-parameter changes. We also tried our MinMax normalization method for DA-RNN, but that also caused accuracy to decrease; we think it's because they tuned their model specifically for the standardization method. We don't believe the different normalization methods are a great advantage for any of the models.

6.2 Baselines

As mentioned in the introduction, the baseline results are taken directly from [Qin et al., 2017]. In that work, they've created 4 baseline models that they use to compare to their DA-RNN architecture. For baselines they tested one deterministic model called ARIMA [Asteriou and Hall, 2011], as

well as 3 RNN-based models: NARX RNN [Diaconescu, 2008], RNN encoder-decoder [Cho et al., 2014b], and RNN encoder-decoder with attention [Bahdanau et al., 2014]. The current SOTA on the SML2010 dataset is the DA-RNN architecture that was the main contribution in [Qin et al., 2017].

The ARIMA (AutoRegressive Integrated Moving Average) model is a generalization of the ARMA model that is popular for non-stationary data [Asteriou and Hall, 2011]. This model generally only uses the target driving series for making predictions.

Most of the RNN based approaches [Qin et al., 2017] tested have been already discussed in chapter 2. NARX RNN [Diaconescu, 2008] is just a classic RNN encoder. The encoder-decoder [Cho et al., 2014b] and attention RNN [Bahdanau et al., 2014] are the same architectures that were used for NMT and discussed in section 2.4.

DA-RNN (Dual Attention RNN) [Qin et al., 2017] is an extension of the standard encoder-decoder with attention. For this model, an input attention mechanism was introduced that filters the input driving series for each encoder time-step. It uses the encoder hidden state at time $t-1$ to compute attention scores a_t^i for each driving series feature x_t^i at time t in order to compute filtered driving series $\hat{x_t^i} = a_t^i \times x_t^i$. This gives DA-RNN's encoder the ability to filter the information from input features for each input time step x^i.

All of the encoder-decoder architectures above use the following process for feeding the input data: the encoder is given *n* steps of the non-target 16 features to encode, and the decoder is fed the *n-1* steps of the target feature that is being predicted.

6.3 Experiments

Metric	Equation
MAE	$\frac{1}{N}\sum_{i=1}^{N} \lvert target^i - X_{pred}^i \rvert$
MAPE	$\frac{1}{N}\sum_{i=1}^{N} \lvert \frac{target^i - X_{pred}^i}{target^i} \rvert \times 100\%$
RMSE	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(target^i - X_{pred}^i)^2}$

Table 6.2: MAE, MAPE, and RMSE metric equations.

We follow the same methodology as [Qin et al., 2017] in order to not introduce bias for our models. We do 10 runs and report the mean and standard deviation for the mean absolute error (MAE), mean absolute percentage error (MAPE), and root mean squared error (RMSE) metrics, for which equations are provided in Table 6.2. MAE and RMSE are two scale-dependent metrics and thus these values should not be compared for different datasets. MAPE on the other hand is a scale-independent metric as it is a percentage and thus it can be used for comparing a model's performance on different datasets. Just like DA-RNN we use the ADAM optimizer with $batch_size = 128$, intake

10 time-steps of data, and train using the mean squared error (MSE) objective function:

$$MSE(X_{pred}, target) = \frac{1}{N}\sum_{i=1}^{N}(target^i - X_{pred}^i)^2 \tag{6.5}$$

where N is the number of training samples in the batch. We test ESSAN with $d_{model} = 64$ and 128; the hyper-parameters used are in Table 6.3. The only thing we change is the amount of heads for multi-head attention in order to preserve d_{head}. Just like DA-RNN, we also decay our learning rate over time. Every 25000 iterations we reduce the learning rate an order of magnitude $(.1)$. For each run we train for 100,000 randomly sampled batches and take the model that got the lowest error cost function value on the validation set as the best model.

The only substantial difference between ESSAN and RNN encoder-decoder based approaches such as DA-RNN is the batch construction. ESSAN has only an encoder module, and thus all of the features for all of the time-steps must be provided together; with the tensor shape being: [batch_size=128, sequence_length=10, num_features=17]. However, the last time step for every sequence cannot contain the value for the target feature which the model is trying to predict. We simply set the value to be the same as the second last target feature value. We've also tried setting this last value to arbitrary values such as 0,-5, and -99, but these approaches led to slightly worse performance.

$\mathbf{d_{model}}$	**Layers**	**Heads**	$\mathbf{d_{head}}$	**L2_Reg**	**l_rate**	**dropout_keep**
64	2	8	16	1×10^{-5}	0.0075	1
128	2	16	16	1×10^{-5}	0.0075	1

Table 6.3: Hyper-parameters used by ESSAN for SML2010 dataset.

Note that we set the dropout_keep value to 1. We've found that any amount of dropout on the input features or self-attention layer lowers the performance. We hypothesize this is because of the small dataset and because too few features are provided for input (unlike sentiment analysis where 300D embeddings were provided). Thus, we only use the l2-regularization as the primary model regularizer.

6.4 Discussion

The time-series predictions results for all of the models discussed can be found in Table 6.4. We report about 2% better metrics compared to the previous SOTA model, DA-RNN. This demonstrates that our fully attention model is also good at sequence modeling for tasks other than NLP. [Qin et al., 2017] showed that most of the performance boost for DA-RNN comes from the feature-wise input attention that is used to filter the input features. We believe that the multi-head attention we use in ESSAN performs a similar 'feature filtering' action, which would explain the on-par performance with DA-RNN.

We did not formally test model efficiency and characteristics differences as we did for our work for sentiment analysis in chapter 5. However, the number of parameters for ESSAN and DA-RNN for $D_{model} = 128$ are 124,064 and 227,067 respectively. This is inline with our findings in section 5.1, and so we expect that the relationship to rest of the characteristics will be similar, with ESSAN being faster to train and perform inference.

Models		SML2010 Dataset		
		MAE ($\times 10^{-2}\%$)	**MAPE** ($\times 10^{-2}\%$)	**RMSE** ($\times 10^{-2}\%$)
Baselines From [Qin et al., 2017]	ARIMA [2011]	1.95	9.29	2.65
	NARX RNN [2008]	1.79±0.07	8.64±0.29	2.34±0.08
	Encoder-Decoder (64) [2014b]	2.59±0.07	12.1±0.34	3.37±0.07
	Encoder-Decoder (128) [2014b]	1.91±0.02	9.00±0.10	2.52±0.04
	Attenion RNN (64) [2014]	1.78±0.03	8.46±0.09	2.32±0.03
	Attenion RNN (128) [2014]	1.77±0.02	8.45±0.09	2.33±0.03
	DA-RNN (64) [2017]	1.53±0.01	7.31±0.05	2.02±0.01
	DA-RNN (128) [2017]	1.50±0.01	7.14±0.07	1.97±0.01
Ours	ESSAN (64)	**1.47±0.04**	**7.03±0.22**	**1.93±0.06**
	ESSAN (128)	1.48±0.05	7.06±0.28	1.95±0.08

Table 6.4: Time series prediction metric results for the SML2010 dataset. The results are mean and standard deviation of 10 runs for the three metrics: MAE, MAPE, and RMSE. The best results for each metric are outlined in **bold**. For each $model(d_{model})$, the dimension of the encoder m and the decoder p is set to $m = p = d_{model} =$ 64 or 128.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have applied various self-attention neural network architectures, some of which are proposed by us, on sentiment analysis classification and time-series prediction regression problems.

We first demonstrated that self-attention is a better building block than recurrence and convolutions by comparing our simple self-attention network (SSAN) with basic RNN LSTM and BiLSTM architectures, as well as a simple convolutional network on six sentiment analysis datasets. We demonstrated that our SSAN architectures performed better than other types of neural networks, all while having more favourable model characteristics such as memory consumption and training/inference times.

Next we proposed an extended SSAN architecture, called ESSAN, that achieved state-of-the-art accuracy on the SST-fine dataset, beating DiSAN [Shen et al., 2018a], the previous best self-attention architecture. Once again, we did that while having more favourable model characteristics such as lower memory consumption, faster training/inference times, and fewer trainable parameters than DiSAN.

Finally, we have applied our ESSAN self-attention architecture on the regression task of time-series prediction, and showed better performance on one dataset than previous best technique, the Dual-Attention RNN (DA-RNN) [Qin et al., 2017] architecture.

7.2 Future Work

We've showed evidence that self-attention networks have significant benefits over previous neural network architectures for classification and regression tasks. However, there's always room for improvement and research to find better sequence modeling methods will continue most likely still for some time.

We plan on testing our SSAN and ESSAN architectures on larger text classification datasets such as Amazon product reviews (3.4m train, 640k test samples) [Zhang et al., 2015], IMDB movie reviews (25k train, 25k test samples) [Maas et al., 2011], and SNLI language inference classification

(570k train samples) [Bowman et al., 2015]. We hypothesize that deeper (3+ SAN layers) ESSAN architecture will be beneficial for datasets containing millions of training samples. In which case similar to the Transformer model, modifying ESSAN to use techniques such as residual connections [He et al., 2016] and layer normalization [Ba et al., 2016] might also be beneficial to help stabilize the training.

We plan to try other methods of creating the final fixed sequence representation vector. Thus far we have done averaging for SSAN and attention method for ESSAN. We plan to try other methods such as max-pooling, feature-level attention such as done by DiSAN, and multi-head attention.

We are also interested in investigating which neural network architecture is better at handling redundant or unimportant features. One of the benefits of DA-RNN and DiSAN is that their attention mechanisms have the ability to perform attention at the feature level. It would be interesting to see how the mentioned models, multi-head attention, as well as RNN and CNN mechanisms perform on such data. We've created a big dataset of 259 features, derived from daily stock market price and volume data. Most of the features in this dataset are technical indicators, such as moving averages, constructed solely from daily Open, High, Low, Close and Volume features. Many of the features are variations of the same technical indicator, thus there is high correlation between various features. In the future, we plan to apply the mentioned models on this dataset to study how each model deals with large amounts of highly correlated driving series.

Finally, due to the highly customizable nature of self-attention networks, we plan to continue the search for the best architecture for sequence modeling tasks. We believe that SOTA results can be achieved on many current datasets by simply finding appropriate combinations of the following components and hyper-parameters: optimizer, dropout rates and locations, L2-regularization rate, activation function, residual connections, layer normalization, learning rate and decay methods, number of self-attention layers, number of heads used for multi-head attention, where to apply feed-forward or projection layers and how deep they should be, etc. For further improvement, we also plan to try ensembling methods for neural network methods, such as these: Checkpoint Ensembling [Chen et al., 2017], Snapshot Ensembling [Huang et al., 2017], Fast Geometric Ensembling [Garipov et al., 2018], Stochastic Weight Averaging [Izmailov et al., 2018].

References

Rodrigo Agerri, Montse Cuadros, Seán Gaines, and German Rigau. Opener: Open polarity enhanced named entity recognition. *Procesamiento del Lenguaje Natural*, 51(0):215–218, 2013. ISSN 1989-7553. URL `http://journal.sepln.org/sepln/ojs/ojs/index.php/pln/article/view/4891`.

Artaches Ambartsoumian and Fred Popowich. Self-attention: A better building block for sentiment analysis neural network classifiers. In *Proceedings of the 9th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*, Brussels, Belgium, November 2018. Association for Computational Linguistics.

Dimitros Asteriou and Stephen G Hall. Arima models and the box-jenkins methodology. *Applied Econometrics*, 2(2):265–286, 2011.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv:1409.0473*, 2014.

Jeremy Barnes, Roman Klinger, and Sabine Schulte im Walde. Assessing state-of-the-art sentiment models on state-of-the-art sentiment datasets. In *Proceedings of the 8th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*, Copenhagen, Denmark, 2017.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

Kanad Chakraborty, Kishan Mehrotra, Chilukuri K Mohan, and Sanjay Ranka. Forecasting the behavior of multivariate time series using neural networks. *Neural networks*, 5(6):961–970, 1992.

Hugh Chen, Scott Lundberg, and Su-In Lee. Checkpoint ensembles: Ensemble methods from a single training process. *arXiv preprint arXiv:1710.03282*, 2017.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *CoRR*, abs/1601.06733, 2016a. URL `http://arxiv.org/abs/1601.06733`.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 551–561, Austin, Texas, November 2016b. Association for Computational Linguistics. URL `https://aclweb.org/anthology/D16-1053`.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical*

Methods in Natural Language Processing (EMNLP), pages 1724–1734. Association for Computational Linguistics, 2014a. doi: 10.3115/v1/D14-1179. URL http://www.aclweb.org/anthology/D14-1179.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv:1406.1078*, 2014b.

François Chollet et al. Keras. https://keras.io, 2015.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Michał Daniluk, Tim Rocktäschel, Johannes Welbl, and Sebastian Riedel. Frustratingly short attention spans in neural language modeling. *arXiv preprint arXiv:1702.04521*, 2017.

Eugen Diaconescu. The use of NARX neural networks to predict chaotic time series. *WSEA Transactions on Computer Research*, 3(3), 2008.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P Vetrov, and Andrew Gordon Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. *arXiv preprint arXiv:1802.10026*, 2018.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

Felix Greaves, Daniel Ramirez-Cano, Christopher Millett, Ara Darzi, and Liam Donaldson. Harnessing the cloud of patient experience: using social media to detect poor quality healthcare. *BMJ Qual Saf*, 22(3):251–255, 2013a.

Felix Greaves, Daniel Ramirez-Cano, Christopher Millett, Ara Darzi, and Liam Donaldson. Use of sentiment analysis for capturing patient experience from free-text comments posted online. *Journal of medical Internet research*, 15(11), 2013b.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Marti A. Hearst. Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28, July 1998. ISSN 1541-1672. doi: 10.1109/5254.708428. URL http://dx.doi.org/10.1109/5254.708428.

Sepp Hochreiter. Gradient flow in recurrent nets : the difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*, pages 237–244, 2001. URL https://ci.nii.ac.jp/naid/10025777304/en/.

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

Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E Hopcroft, and Kilian Q Weinberger. Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109*, 2017.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*, 2018.

Bernard J Jansen, Mimi Zhang, Kate Sobel, and Abdur Chowdury. Twitter power: Tweets as electronic word of mouth. *Journal of the American society for information science and technology*, 60(11):2169–2188, 2009.

Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 655–665, Baltimore, Maryland, June 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P14-1062`.

Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aäron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *CoRR*, abs/1610.10099, 2016. URL `http://arxiv.org/abs/1610.10099`.

Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, May 2015. URL `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014a. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1181`.

Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751. Association for Computational Linguistics, 2014b. doi: 10.3115/v1/D14-1181. URL `http://www.aclweb.org/anthology/D14-1181`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327, 2016. ISSN 2307-387X. URL `https://transacl.org/ojs/index.php/tacl/article/view/885`.

Filippos Kokkinos and Alexandros Potamianos. Structural attention neural networks for improved sentiment analysis. *arXiv preprint arXiv:1701.01811*, 2017.

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

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

Tao Lei, Regina Barzilay, and Tommi Jaakkola. Molding cnns for text: non-linear, non-consecutive convolutions. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1565–1575. Association for Computational Linguistics, 2015. doi: 10.18653/v1/D15-1180. URL http://www.aclweb.org/anthology/D15-1180.

Jiwei Li, Thang Luong, Dan Jurafsky, and Eduard Hovy. When are tree structures necessary for deep learning of representations? In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2304–2314. Association for Computational Linguistics, 2015. doi: 10.18653/v1/D15-1278. URL http://www.aclweb.org/anthology/D15-1278.

Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018.

Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. Hierarchical question-image co-attention for visual question answering. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 289–297. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6202-hierarchical-question-image-co-attention-for-visual-question-answering.pdf.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

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

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

Mika V. Mäntylä, Daniel Graziotin, and Miikka Kuutila. The evolution of sentiment analysis—a review of research topics, venues, and top cited papers. *Computer Science Review*, 27:16 – 32, 2018. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2017.10.002. URL http://www.sciencedirect.com/science/article/pii/S1574013717300606.

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

Preslav Nakov, Zornitsa Kozareva, Alan Ritter, Sara Rosenthal, Veselin Stoyanov, and Theresa Wilson. Semeval-2013 task 2: Sentiment analysis in twitter, 2013.

Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Caglar Gulcehre, and Bing Xiang. Abstractive text summarization using sequence-to-sequence rnns and beyond. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 280–290, Berlin, Germany, August 2016. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/K16-1028`.

Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Found. Trends Inf. Retr.*, 2(1-2): 1–135, January 2008. ISSN 1554-0669. doi: 10.1561/1500000011. URL `http://dx.doi.org/10.1561/1500000011`.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Łukasz Kaiser, Noam Shazeer, and Alexander Ku. Image transformer. *arXiv preprint arXiv:1802.05751*, 2018.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`.

Barbara Plank, Anders Søgaard, and Yoav Goldberg. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 412–418, Berlin, Germany, August 2016. Association for Computational Linguistics. URL `http://anthology.aclweb.org/P16-2067`.

Daniel Povey, Hossein Hadian, Pegah Ghahremani, Ke Li, and Sanjeev Khudanpur. A time-restricted self-attention layer for asr. 2017.

Qiao Qian, Minlie Huang, Jinhao Lei, and Xiaoyan Zhu. Linguistically regularized lstm for sentiment classification. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1679–1689, Vancouver, Canada, July 2017. Association for Computational Linguistics. URL `http://aclweb.org/anthology/P17-1154`.

Yao Qin, Dongjin Song, Haifeng Chen, Wei Cheng, Guofei Jiang, and Garrison W. Cottrell. A dual-stage attention-based recurrent neural network for time series prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 2627–2633, 2017. doi: 10.24963/ijcai.2017/366. URL `https://doi.org/10.24963/ijcai.2017/366`.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017. URL `http://arxiv.org/abs/1710.05941`.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(9):533–536, 1986.

Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *EMNLP*, 2015.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N18-2074`.

T. Shen, T. Zhou, G. Long, J. Jiang, and C. Zhang. Fast Directional Self-Attention Mechanism. *ArXiv e-prints*, May 2018.

Tao Shen, Tianyi Zhou, Guodong Long, Jing Jiang, Shirui Pan, and Chengqi Zhang. Disan: Directional self-attention network for rnn/cnn-free language understanding. In *AAAI Conference on Artificial Intelligence*, 2018a.

Tao Shen, Tianyi Zhou, Guodong Long, Jing Jiang, and Chengqi Zhang. Bi-directional block self-attention for fast and memory-efficient sequence modeling. *CoRR*, abs/1804.00857, 2018b.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

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

Emma Strubell, Patrick Verga, Daniel Andor, David Weiss, and Andrew McCallum. Linguistically-informed self-attention for semantic role labeling. *CoRR*, abs/1804.08199, 2018.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf`.

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

Zhiyang Teng, Duy Tin Vo, and Yue Zhang. Context-sensitive lexicon features for neural sentiment analysis. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1629–1638, Austin, Texas, November 2016. Association for Computational Linguistics. URL `https://aclweb.org/anthology/D16-1169`.

Peter D Turney. Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 417–424. Association for Computational Linguistics, 2002.

Olga Uryupina, Barbara Plank, Aliaksei Severyn, Agata Rotondi, and Alessandro Moschitti. Sentube: A corpus for sentiment analysis on youtube social media. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation*. European Language Resources Association, 2014.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia 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 30*, pages 5998–6008. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf`.

Oriol Vinyals and Quoc Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.

Xingyou Wang, Weijie Jiang, and Zhiyong Luo. Combination of convolutional and recurrent neural network for sentiment analysis of short texts. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 2428–2437, 2016.

Andreas S Weigend. *Time series prediction: forecasting the future and understanding the past*. Routledge, 2018.

Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Ronald J Williams and David Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. *Backpropagation: Theory, architectures, and applications*, 1: 433–486.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *arXiv preprint arXiv:1708.02709*, 2017.

Fransisco Zamora-Martínez, Pablo Romeu, Pablo Botella-Rocamora, and Juan Pardo. On-line learning of indoor temperature forecasting models towards energy efficiency. *Energy and Buildings*, 83:162–172, 2014.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. *arXiv preprint arXiv:1805.08318*, 2018.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

Peng Zhou, Zhenyu Qi, Suncong Zheng, Jiaming Xu, Hongyun Bao, and Bo Xu. Text classification improved by integrating bidirectional lstm with two-dimensional max pooling. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3485–3495, Osaka, Japan, December 2016. The COLING 2016 Organizing Committee. URL http://aclweb.org/anthology/C16-1329.

Yi Zhou, Junying Zhou, Lu Liu, Jiangtao Feng, Haoyuan Peng, and Xiaoqing Zheng. Rnn-based sequence-preserved attention for dependency parsing. 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17176.