University of Nottingham
UK | CHINA | MALAYSIA

Conversational AI Chatbot
G53IDS Dissertation

Author
Student ID: 14307825

Supervisor
Dr. Ke Zhou

I hereby declare that this dissertation is all my own work, except as indicated in the text.

School of Computer Science
University of Nottingham

15th May 2020

Abstract

Conversational artificial intelligence is still considered to be in its infancy with many companies and universities investing heavily in this area of AI. This is because currently Although great progress has already been made with commercial social-bots such as Siri and Amazon Alexa, no system is close to passing the turing test. This is because conversational chatbots suffer from a number of issues. One of them being the inability to keep a consistent persona expressed as a set of backgrounds facts.

For example, a social-bot is likely to give contradicting answers to the same question asked twice, if the question is paraphrased the second time. The main goal is to minimize this behaviour as much as possible by using machine learning models which are variations of the Recurrent Neural Network (RNN) and other novel models.

Acknowledgements

I would like to thank my supervisor Dr. Ke Zhou and Wen Zheng for their help and guidance during this project. I would also like to extend my gratitude to my personal tutor Dr. Jason Atkin, Riyadh and Feras for supporting me during a tough time.

Contents

1 Background and Motivation

The global Natural Language Processing (NLP) market value is predicted to rise from $10.2 billion in 2019 to $26.4 billion in 2024 with a Compound Annual Growth Rate of 21% during this time [23]. Conversational AI is responsible for a sizeable chunk of this market value. This is because many companies are now looking to automate many of the human to human interactions which take place between employees and customers. For example, airlines are guiding customers through the process of booking flights or editing reservations via chatbots. This in turn removes the need for customer support staff which increases profits. Many companies are realising the drastic benefits behind incorporating conversational AI into their business model. However, improvements need to made to these chatbots in order to make them viable in a real world production setting.

Having said this, conversational AI has advanced significantly in recent years with the rise of virtual assistants such as Siri and Amazon Alexa. Amazon has invested heavily into the advancement of conversational AI with the establishment of the Alexa Prize [1]. A competition where research groups are given a grant of $250,000 to create a realistic chatbot which is capable of outperforming Alexa in its current state, with the top 3 groups winning a larger grant and inclusion of their work into the next version of Alexa.

Nonetheless, conversational AI is still in it's infancy and there are three main problems which social-bots face today:

- Lack of consistent persona [17]. This means social-bots do not maintain a set of background facts about themselves and are likely to contradict themselves even within a single short conversation.
- Lack of long term memory with respect to the current conversation as models are usually trained to produce an utterance based mostly on the recent dialogue [40]. This means the social-bot is likely to forget information earlier in the conversation leading to unnatural interactions with humans.
- Predisposition to answer questions with non-specific answers such as "I'm not sure" [16].

Each of these individually are hard to solve problems in their own respect. This project will be focused on solving the problem of a lack of persona. The overall goal is to create a social-bot which can participate in a non-goal oriented dialogue while maintaining a consistent persona. Concretely, a persona can be defined as a set of background facts which are remembered and used during conversation. When asking a chatbot multiple paraphrased questions the responses may show a lack of consistency. This occurs because the social-bot will reflect the persona of the responses within the dataset it was trained on, but these responses come from many different people with different personalities and background facts.

message	Where do you live now?
response	I live in Los Angeles.
message	In which city do you live now?
response	I live in Madrid.
message	In which country do you live now?
response	England, you?
message	Where were you born?
response	I was born in Canada.
message	Where are you from?
response	England, you?
message	Where did you grow up?
response	I grew up in Texas.
message	How old are you?
response	16 and you?
message	What's your age?
response	18.
message	What is your major?
response	I'm majoring in psychology
message	What did you study in college?
response	English lit.

Figure 1: Example of inconsistent persona generated by a "4-layer SEQ2SEQ model trained on 25 million Twitter conversation snippets" [17].

The lack of a consistent and coherent persona makes it impossible for social-bots in their current state to pass the turing test and reduces the quality of chatbots deployed in the real world resulting in lower customer satisfaction. This is because some of the answers the social-bot gives will be obvious contradictions to previous answers making the distinction between human and chatbot clear. The key objectives of this project are to:

- Implement and train a Sequence to Sequence Neural Network architecture [36] to allow the social-bot to map input questions of varying lengths to meaningful answers as a base model.
- Implement Attention [3] on top of the Sequence to Sequence model.
- Implement and train a new type of model based on the Sequence to Sequence architecture which explicitly takes into account the chatbots given persona.
- Implement and train the Transformer model [39], a model based solely on Dot-Product attention [21].
- Use GloVe word embeddings [26] in order to convert words into fixed length vectors.
- Compare the effectiveness of the trained models by using automatic and human evaluation metrics.

2 Related Work

Ever since electronic computers were invented, programmers and researchers alike have been interested in the idea of creating a thinking machine, which is capable of conversing with a human successfully. Alan Turing even made this the benchmark of Artificial Intelligence with his conception of the Turing test [38].

2.1 Hard Coded Rules

The first attempts at cracking this problem involved writing thousands of handcrafted rules to generate an output sentence based on the input sentence. Essentially, this boils down to splitting the input sentence into words and using many if statements to determine what the output should be.

This approach did achieve some mediocre results, programs using this methodology were sometimes able to produce meaningful responses to input sentences but the quality of responses were not good. ELIZA was the first example of such a program [41].

Overall this approach is simple but was not effective because generated responses were very predictable. On top of this, source code for such programs could easily reach the tens of thousands in lines of code. This makes writing such a program laborious and in-efficient because these rules must be handcrafted in accordance with each other and added manually to the source code.

2.2 Statistical Natural Language Processing

From the early 1980's researchers realised better results could be achieved by treating the text generation problem as a machine translation problem [30]. That is given an input sentence and the output words generated so far, what is the probability distribution over all words in the vocabulary.

$$p(y^{<t>}|y^{<1>}, ..., y^{<t-1>}, X) \tag{1}$$

This distribution is calculated through a complicated pipeline of statistical models and heuristics. For a time, Statistical Machine Translation (SMT) models were considered state-of-the-art. The problem with these models is they require a lot of specialized knowledge in linguistics and they are not end-to-end models. What this means is that a model is based on several different subsystems which were individually hand crafted using human knowledge about morphology, syntax, semantics and pragmatics as well as general knowledge about the world. Another issue with these class of models is that their performance plateaus even when more data is given to them.

2.3 Neural Natural Language Processing

This approach involves training end-to-end deep learning models which can solve the problem of text generation or machine translation. Only a single model is required to perform the task instead of a pipeline of models. This approach relies on the fact that neural models have the capability to be trained over a large dataset to learn for themselves how to solve the problem.

Only recently, has this approach become viable because of the exponential growth of computing power due to Moore's law [10]. At the present day Neural Natural Language Processing, for example Neural Machine Translation (NMT) is the state-of-the-art and looks to be a clear winner over the other methodologies. This is becuase of the following reasons:

- **Automatic Feature Learning:** Models not using deep learning have manually designed features which tend to be over specified, incomplete, take a long time to design, validate, and cannot exceed a ceiling level of mediocre performance. Whereas, deep learning models will automatically find which features to focus on within the data and can find useful patterns for prediction automatically without the need for specialised knowledge of linguistics.
- **No Plateau:** Deep learning models do not seem to plateau in performance unlike statistical models. Deep learning models are the only known class of algorithms which consistently gain better results when fed more data. There doesn't seem to be a plateau to the results they can achieve with respect to the amount of training data they receive. This is a huge benefit, especially today where there is an abundance of data and processing power has been increasing at an exponential rate.
- **Continued Development:** Over the last 6 years improvement and development of deep learning models for Natural Language Processing has been increasing at an unprecedented rate. Currently, all state-of-the-art models use deep learning and every year there is consistent improvements to these models.

A lot of these powerful neural models, for example Sequence to Sequence or Transformers are readily tested on common problems such as Machine Translation, but in this project we wish to compare their effectiveness when it comes to keeping a consistent persona while generating meaningful responses, a unique problem seldom explored. We also wish to adapt these models and develop novel models for the specific problem at hand.

3 Methodology

3.1 Artificial Neural Networks

Artificial Neural Networks (ANN) also known as Multi-layer Perceptrons (MLP) are the most fundamental type of neural networks. They are an old concept conceived by Frank Rosenblatt in the late 1950's [32]. The MLP takes inspiration from the human brain and was originally designed to be a mathematical model to perform the same computations as interconnected neurons within the brain.

3.1.1 Architecture

ANN's are made up of layers of neurons. Every neuron within a layer including a bias unit is connected to every neuron in the next layer with each of these individual connections having an associated weight. A single neuron is simply a weighted sum of all the neurons in the previous layer, followed by a non-linear activation function of this weighted sum.

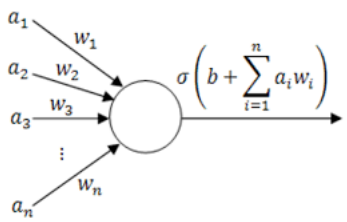

Figure 2: A single neuron/perceptron/node in an ANN

Non-linear activation functions are a critical part of the MLP because without them the output of a MLP will only ever be a linear combination of the input features, no matter the number of hidden layers used. Therefore, if the input data is linearly non-separable, the MLP will fail to classify data samples with any success because linearly non-seperable data can not be separated with a linear combination of the inputs by definition.

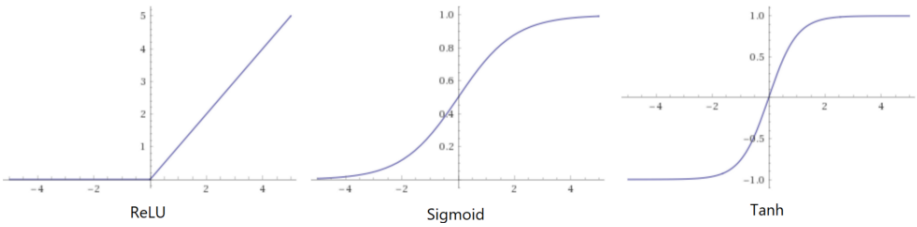

Figure 3: Common activation functions used within ANN's

The first layer in a MLP, also called the input layer is the features which the model will take as input. Hidden layers are subsequent layers and the final layer is the output of the model. The number of neurons in the output layer is dependant on the problem. If only one output is required, for example a regression or two class classification problem, then naturally only one node in the final layer is required.

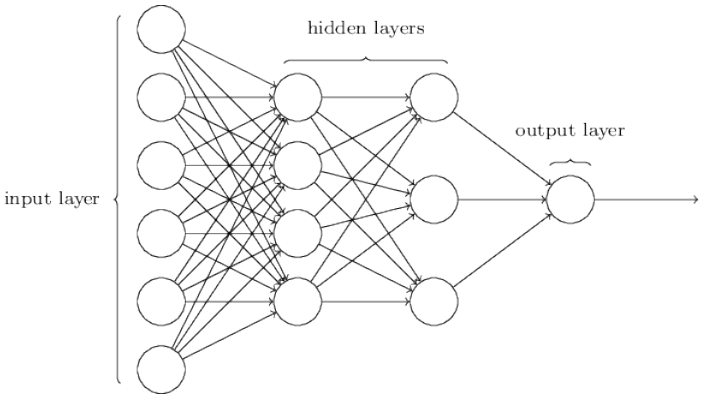

Figure 4: A MLP with two hidden layers and a single output node.

The intuition behind why this type of model works is that the connection between the final hidden layer and output layer is equivalent to logistic regression.

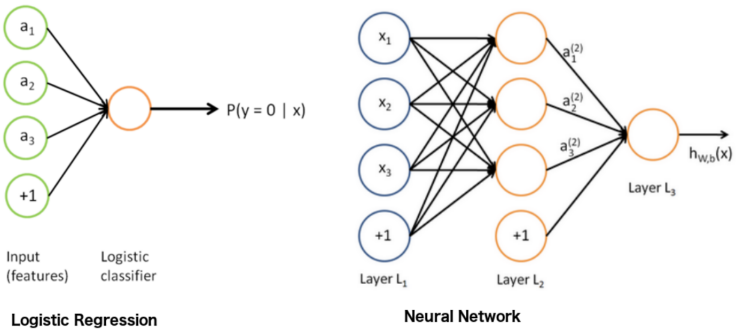

Figure 5: Logistic Regression vs ANN. Logistic regression is equivalent to an ANN which has no hidden layers.

Logistic regression will classify linearly separable data with high accuracy. For many problems the input feature space is linearly non-separable, however the MLP will learn its own features in the hidden layers. If these learnt features, whatever they may be are within a linearly separable vector space at the final hidden layer then the MLP will be able to classify examples with a high accuracy because this will be equivalent to performing logistic regression on linearly separable data. The architecture of an ANN refers to the number of hidden layers and how many nodes are within

each of these layers. It's not clear exactly what this architecture should be and the optimal architecture changes from problem to problem. Therefore, it should be tuned as a hyper parameter by training different architectures and choosing that which performs best on a validation set.

As stated in the beginning of this section, MLP's are not necessarily a new invention, however only relatively recently have they been used to solve real world problems because of significant increases in computing power and readily available large scale datasets, which are required to train such models effectively.

3.1.2 Forward Propagation

Forward Propagation is the process of feeding an ANN with a set of real valued inputs and propagating this input through the various hidden layers one by one until the output layer is reached, in which case this will be the output of the model.

The ANN's weights to be trained are a set of real valued matrices, where each each layer except the output layer has an associated weight matrix. To calculate the values of the nodes in the first hidden layer $l_2 \in \mathbb{R}^{m\times 1}$ given the values of the input nodes $l_1 \in \mathbb{R}^{n\times 1}$, the weight matrix $w_1 \in \mathbb{R}^{m\times n}$ and the bias for this layer $b_1 \in \mathbb{R}^{m\times 1}$, we can perform the following operation:

$$l_2 = g(w_1 l_1 + b_1) \tag{2}$$

Note that here g is an non-linear activation function applied element wise. This formula can be extrapolated to calculate the value of nodes in any layer l_{j+1} given the values of the nodes in the previous layer l_j. Therefore, the output of the MLP is determined by calculating the values of each layer one by one using a for loop, starting from the layer directly after the input layer.

$$\begin{gathered} l_{j+1} = g(w_j l_j \ + \ b_j) \quad for\ j = 1, ..., L-1 \\ where\ L\ is\ the\ number\ of\ layers \end{gathered} \tag{3}$$

Currently, this will calculate the output for a single data point which is simply a single row vector. Instead of performing the above explicitly in a for loop throughout all the available data points to get their outputs, we can use a vectorised implementation, which will be more efficient since it will be calculating the output for all training examples in a single set of matrix multiplications.

A vectorised implementation is achieved simply by performing the same calculation above, but instead of feeding the network with an input $l_1 \in \mathbb{R}^{n\times 1}$, we feed in a matrix $l_1 \in \mathbb{R}^{n\times k}$ where k is the number of data points to forward propagate and n are the number of features for each data point. k is typically the number of data points in the whole training set or a preset batch size if the whole training set will not fit into main memory.

3.1.3 Back Propagation

The weights and biases seen in the previous sub-section are randomly initialised, but they need to be trained in order to achieve optimal results in terms of the predictions which the model outputs.

This happens through Gradient Descent [31]. This algorithm underpins all neural models. It works by defining a loss function suitable for the problem at hand and then using the chain rule to find the partial derivatives of the chosen loss function with respect to the weights throughout the model. We wish to minimise the loss function as this will result in better predictions. The derivative will tell us the slope of the loss function and therefore, the direction to move in for each weight individually. In order to find the global minimum of the loss function, all we have to do is nudge each weight individually by a value proportional to the negative derivative of the loss function with respect to that weight over a number of iterations until the derivatives are close to 0 or for a preset number of iterations also called epochs.

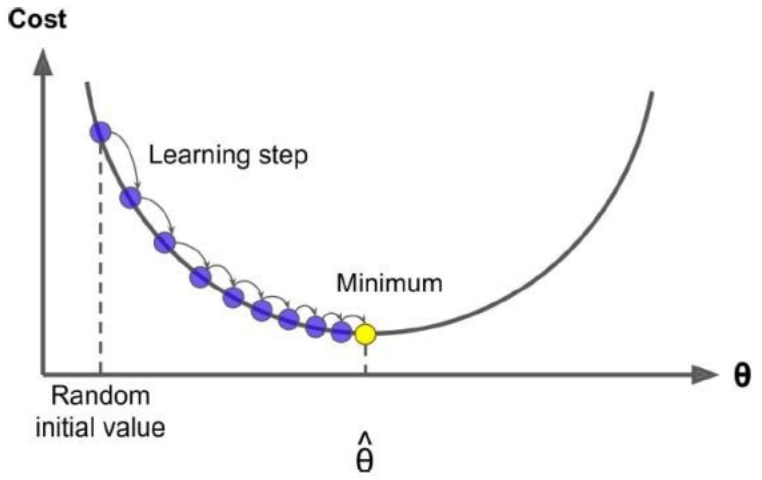

Figure 6: Visualisation of Gradient Descent over a number of iterations, considering a single weight value θ.

Concretely, this can be achieved by iterating over the following operation until convergence of the weights:

$$W_j := W_j - \alpha \frac{\partial}{\partial W_j} J(W) \quad for\ j = 1, ..., |W| \tag{4}$$

Note that here W is a row vector containing all the weights in the model. Biases are treated to be exactly the same as weights. α also known as the learning rate is a hyper-parameter to be tuned and is usually in the range $0 < \alpha < 1$ and J is the loss function.

The first step is choosing a suitable activation function for the nodes in the output layer and identifying the number of nodes in the output layer as this will determine which cost function to use. In our case we wish to learn a probability distribution over all the words in the vocabulary and then choose the word with the highest probability. Therefore, the number of nodes in the output layer should be equal to the vocabulary size V. All output nodes must represent a probability within a larger probability distribution which sums to 1.0. The softmax activation function is perfect for this:

$$softmax(x)_i = \frac{exp(x_i)}{\sum_{j=1}^{|x|} exp(x_j)} \tag{5}$$

Here $x \in \mathbb{R}^{V \times 1}$ is a row vector representing the output layer of the network and therefore, the output of the softmax is a row vector with the same dimensionality.

The next step is to choose a suitable cost function. Our probability distribution should be highest at the ground truth of the word to output, which is given by the label for the current training example during training since we assume a labelled dataset. Essentially, this is equivalent to a multi-class classification problem. Categorical Cross Entropy will penalise the model based on how far away the output node corresponding to the ground truth is to a probability of 1.0. This is done using log likelihood probability.

$$J(W) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{V} y_{i,j} \log(\hat{y}_{i,j}) \tag{6}$$

Note that here m is the number of data points in the training set or the batch size, depending on which was used during forward propagation, $y \in \mathbb{R}^{V \times m}$ is a matrix containing the one-hot encoded labels for each training example stacked horizontally and $\hat{y} \in \mathbb{R}^{V \times m}$ are the outputs of the model produced by forward propagation.

At this point we are almost ready to run Gradient Descent, which will let the model learn the optimal weights. The last thing remaining is to calculate $\frac{\partial}{\partial W_j} J(W)$ for each individual weight. This is where back propagation [33] is used, it automatically calculates these partial derivatives by back propagating error terms from the output nodes to the input nodes in accordance to the chain rule. Below is a simple example of the chain rule and how it can be used

to differentiate functions of functions, which is exactly what a MLP is.

$$\begin{gathered} y = \cos x^2 \\ u = x^2 \\ y = \cos u \\ \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}u} \cdot \frac{\mathrm{d}u}{\mathrm{d}x} \\ \frac{\mathrm{d}y}{\mathrm{d}x} = -2x \sin x^2 \end{gathered} \tag{7}$$

By virtue of the chain rule, the partial derivative of the cost function with respect to (w.r.t.) an individual weight $w_{l,j,k}$ is given by:

$$\frac{\partial}{\partial w_{l,j,k}} J(W) = \frac{\partial}{\partial a_{l+1,j}} J(W) \cdot \frac{\partial a_{l+1,j}}{\partial z_{l+1,j}} \cdot \frac{\partial z_{l+1,j}}{\partial w_{l,j,k}} \tag{8}$$

Note that here $w_{l,j,k}$ is the weight connection between the k^{th} node in layer l to the j^{th} node in layer $l+1$. $a_{l,j}$ is the value of the j^{th} node at layer l after passing through that nodes non-linear activation function. $z_{l,j}$ is just the weighted sum of the j^{th} node at layer l before the non-linear activation function is applied.

Back propagation introduces an error term $\delta_{l,j}$ associated with each node in the network which is simply the partial derivative of the cost function w.r.t. the weighted sum at that node.

$$\delta_{l,j} = \frac{\partial}{\partial z_{l,j}} J(W) = \frac{\partial}{\partial a_{l,j}} J(W) \cdot \frac{\partial a_{l,j}}{\partial z_{l,j}} \tag{9}$$

So we can simplify equation (8) to the following:

$$\frac{\partial}{\partial w_{l,j,k}} J(W) = \delta_{l+1,j} \cdot \frac{\partial z_{l+1,j}}{\partial w_{l,j,k}} \tag{10}$$

The partial derivative of the weighted sum of node j at layer $l+1$ w.r.t. a weight directly included in that weighted sum, which is given by $\frac{\partial z_{l+1,j}}{\partial w_{l,j,k}}$, is nothing more than the the value of node k at layer l after passing through it's activation function. This is the term $a_{l,k}$. Therefore, equation (10) can be simplified to the following:

$$\frac{\partial}{\partial w_{l,j,k}} J(W) = \delta_{l+1,j} \cdot a_{l,k} \tag{11}$$

So if we can calculate the error term $\delta_{l,j}$ and activation $a_{l,k}$ for every node in the ANN, then we will have the partial derivatives that are required. The activations $a_{l,k}$ are trivially calculated and stored during forward propagation. Error terms at the final layer L are calculated first as these are closest to the error function and therefore require the least number of chain rule steps. Assuming Categorical Cross Entropy as the cost function and the use of the softmax function at the output layer, we have:

$$\frac{\partial}{\partial a_{L,j}} J(W) = \frac{-y_j}{a_{L,j}} + \frac{1 - y_j}{1 - a_{L,j}} \tag{12}$$

$$= \frac{a_{L,j} - y_j}{a_{L,j}(1 - a_{L,j})} \tag{13}$$

$$\frac{\partial a_{L,j}}{\partial z_{L,j}} = a_{L,j}(1 - a_{L,j}) \tag{14}$$

$$\delta_{L,j} = a_{L,j} - y_j \tag{15}$$

Note that $y \in \mathbb{R}^{V \times 1}$ is a one-hot encoded label vector containing the ground truth for the current training example.

Once the error terms in the final layer are calculated, they can be back propagated in a very similar fashion to forward propagation, except traversing the MLP in reverse and using the error terms as the values to multiply with the weights. This allows us to calculate all the error terms for each node in the network. This works because it saves us from performing repeated chain rule calculations for non-output layers. Below is a formula for a vectorised implementation of calculating all the error terms:

$$\delta_L = a_L - y \tag{16}$$

$$\delta_l = ((w_l)^\top \delta_{l+1}) \odot a_l \odot (1 - a_1) \quad for\ l = L - 1, ..., 2 \tag{17}$$

Here we assume sigmoid activation function for all nodes not in the output layer, which is where the element-wise multiplication term $a_l \odot (1 - a_1)$ comes from as this is the derivative of the sigmoid function w.r.t. it's inputs, as shown below:

$$g(x) = \frac{1}{1 + exp(-x)} \tag{18}$$

$$g'(x) = g(x)((1 - g(x)) \tag{19}$$

To conclude an ANN is able to learn its weight by first performing forward propagation in order to store the activations at each node. Then the error terms at the final layer are calculated by a simple subtraction between the outputs and labels. After this, all the subsequent error terms are calculated using back propagation.

These error terms are then multiplied by their corresponding activation values at the previous layer, in which case we have the derivatives for all the weights in the network for a single training example. The cost function is summed over m, which is the size of the training set, so we sum the derivative of each weight individually over all training examples or a preset batch size, if the whole training set is too large to fit into memory. We can then perform a single iteration of Gradient Descent. This process is then repeated for a number of epochs or until convergence.

3.1.4 Artificial Neural Networks Lack Temporality

The objective is to create a non-goal oriented chit-chat dialogue system. This is a supervised learning problem where each training example is a response to an utterance. The words in the input sequence are the features and the words in the response sequence are the labels. Our model should learn to output an appropriate reply to an input utterance with consistent persona by using a large text corpus of conversations.

Deep Artificial Neural Networks (DNN) [34] are very powerful and have been used to solve a huge range of problems successfully, especially for classification problems, for example image recognition or spam classification. However, DNN's require a fixed vector input because the architecture of DNN's must be fixed and the number of input nodes cannot change unless a new DNN is created. This means DNN's cannot be used to solve this type of Natural Language Processing problem because each input utterances naturally contains a different number of words and therefore, a fixed architecture is not ideal for this problem.

On top of this, DNN's do not have a temporal dimension which is to say each prediction produced by the DNN does not influence subsequent predictions. This property is not useful for problems such as image classification because each image is separate and has no influence on the classification of subsequent images. However, for problems linked to a time series or a sequence of predictions, allowing predictions to influence subsequent predictions is an important condition for success. For example, a DNN which takes a single word represented by a one-hot encoded vector over the vocabulary as input and then produces a single word as output will not identify the difference between the following two sentences:

- "The neural network made a man"
- "The man made a neural network"

This is because each word must be fed separately into the DNN and therefore, the DNN has no sense of the order of these words, even though grammatically the order of words in a sentence is critical towards forming the underlying meaning behind the sentence.

Furthermore, there's the problem that for each input word, the DNN can only produce a single word as output meaning the number of words in the reply produced by the DNN would have to be equal to the number of words in the input sentence. Real life replies to a sentence are rarely the same length in words as the given sentence. All these factors make the DNN unsuited to the problem at hand. Recurrent Neural Networks (RNN) are an an adapted version of DNN's to allow a temporal dimension to be included within inputs.

3.2 Recurrent Neural Networks

The Recurrent Neural Network (RNN) [19] does not suffer from the issues described in the previous section. Namely, they allow previous predictions to influence future predictions for a single time series input. In our case they take an input sentence one token at a time and produce an output on each token, where each word is a token which could be represented in a number of ways, for example a one-hot vector over all words in the vocabulary or word embeddings.

3.2.1 Architecture

RNN's work by having two inputs: an activation vector also called the hidden state which is passed from the previous timestep and the input token for the current timestep. It will then produce the hidden state for the next timestep and an output for the current timestep.

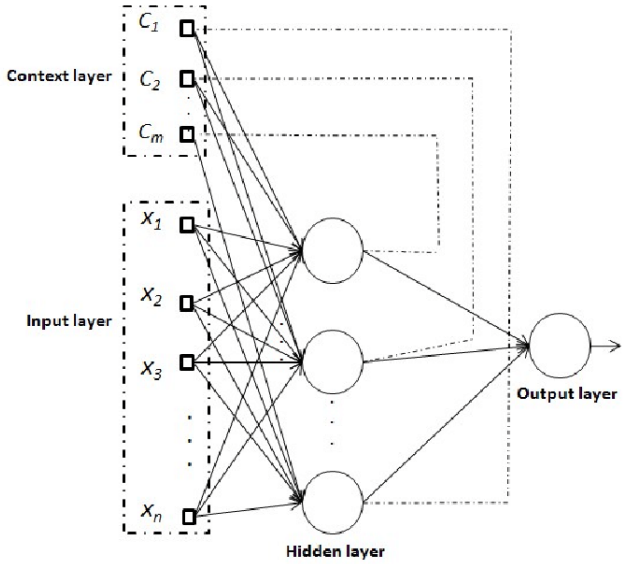

Figure 7: The architecture of a RNN. It has a single hidden layer. The outputs of the hidden layer at timestep t are concatenated with the inputs at timestep $t+1$. Using this mechanism, an RNN uses previous predictions to influence future predictions. Note here the context layer, activation vector or hidden state are interchangeable terminology.

3.2.2 Forward Propagation

Concretely, the RNN will iterate over the following equations for timesteps $t = 1, ..., T$ where T is the length of the current input time series:

$$h^{<t>} = g(w_{hh}h^{<t-1>} + w_{hx}x^{<t>} + b_h) \tag{20}$$

$$\hat{y}^{<t>} = h(w_{yh}h^{<t>} + b_y) \tag{21}$$

The matrices and biases $w_{hh}, w_{hx}w_y, b_h, b_y$ are learned through the Back Propagation Through Time (BPTT) algorithm [42]. g and h are a non-linear activation functions.

The output $\hat{y}^{<t>}$ as well as the input $x^{<t>}$ can be optionally removed at specific timesteps, depending on the problem being solved. A many-to-many RNN will use the output $\hat{y}^{<t>}$ for each timestep. Many-to-many RNN's can be used for image classification using predictions from previous frames and the current frame from a video. One-to-one RNN's have no recurrence and could be used for classifying single images. One-to-many RNN's takes an input only at the first timestep and produce an output for every timestep. This could be used for image captioning where a single image is taken as input, but multiple words are produced as output. Many-to-one RNN's only produce an output at the final timestep. This is useful for sentiment analysis.

So far, none of these types of RNN's can produce an output sequence of a variable length, which is not directly reducible from the length of the input sequence. This is what we need to map input sentences to replies. To solve

this problem another type of many-to-many RNN's called Sequence to Sequence models [36] are used. These are discussed in more detail later.

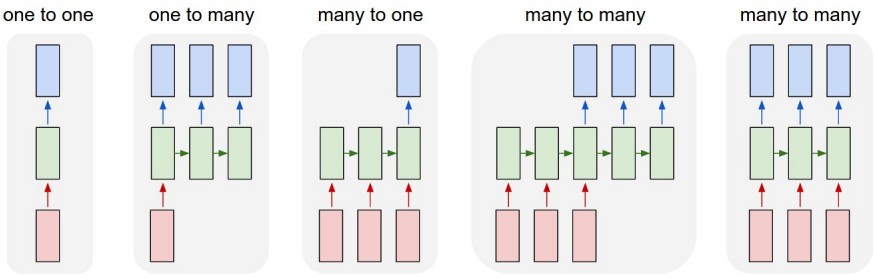

Figure 8: Different ways of configuring a RNN based on it's inputs and outputs.

3.2.3 Back Propagation Through Time

Back Propagation Through Time (BPTT) [42] is an algorithm to calculate the derivatives of a modified cost function w.r.t. the weights and biases $w_{hh}, w_{hx}, w_{yh}\ b_a, b_y$. These derivatives can then used within gradient descent in order to learn the optimal weights.

The RNN will produce a prediction for each timestep. Therefore, each prediction at timestep t needs to be considered within the cost function. Below is Categorical Cross Entropy over a time series:

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{t=1}^{T_y}\sum_{j=1}^{V} y_{i,j}^{<t>}\log(\hat{y}_{i,j}^{<t>}) \tag{22}$$

the cost function $J(W)$ can differentiated w.r.t. the individual weights using the chain rule. Similar to back propagation, calculating already calculated derivatives is prevented by introducing an error term for each node in the output layer $\delta_{3,j}$ at each timestep.

$$\delta_{3,j}^{<t>} = \frac{\partial}{\partial \hat{y}_j^{<t>}} J(W) \cdot \frac{\hat{y}_j^{<t>}}{\partial z_{3,j}^{<t>}} \tag{23}$$

$$= \frac{\partial}{\partial \hat{y}_j^{<t>}} J(W) \cdot h'(z_{3,j}^{<t>}) \tag{24}$$

Assuming the activation function h at the output layer is a softmax:

$$\delta_{3,j}^{<t>} = \hat{y}_j^{<t>} - y_j^{<t>} \tag{25}$$

We also need the error term for each node in the hidden layer $\delta_{2,j}$, for each timestep. This is calculated by not only back propagating error terms from the output nodes at the current timestep t, but also error terms from the hidden layer at timestep $t+1$. Essentially, this is just traversing the RNN in reverse from timesteps $T, ..., 1$ where $\delta_{2,j}^{<T+1>} = 0$. Therefore, we have:

$$\delta_{2,j}^{<t>} = g'(z_{2,j}^{<t>})\left(\sum_{k=1}^{V}\delta_{3,k}^{<t>} w_{k,j}^{yh} + \sum_{k=1}^{h_s}\delta_{2,k}^{<t+1>} w_{k,j}^{hh}\right) \tag{26}$$

Where V are the number of nodes in the output layer, in our case the vocabulary size. h_s are the number of nodes in the hidden layer, $z_{2,j}^{<t>}$ is the weighted sum before non-linearity at the hidden layer (layer 2) at node j at timestep t and $w_{k,j}^{yh}$ is the weight connection between the k^{th} node in the output layer and the j^{th} node in the hidden layer.

Now that we have the error terms for each layer, at each timestep, we can find the derivative of each individual weight at each timestep. We then sum the derivatives over all timesteps since the same weights are shared across all timesteps. Derivatives for the weights connected from the hidden state nodes to the output nodes:

$$\frac{\partial}{\partial w_{j,k}^{yh}} J(W) = \sum_{t=1}^{T} \frac{\partial}{\partial z_{3,j}^{<t>}} J(W) \cdot \frac{\partial z_{3,j}^{<t>}}{\partial w_{j,k}^{yh}} \tag{27}$$

$$= \sum_{t=1}^{T} \delta_{3,j}^{<t>} \cdot h_k^{<t>} \tag{28}$$

Derivatives for the weights connected to the hidden state nodes at the previous timestep to the hidden state nodes at the current timestep nodes:

$$\frac{\partial}{\partial w_{j,k}^{hh}} J(W) = \sum_{t=1}^{T} \frac{\partial}{\partial z_{2,j}^{<t>}} J(W) \cdot \frac{\partial z_{2,j}^{<t>}}{\partial w_{j,k}^{hh}} \tag{29}$$

$$= \sum_{t=1}^{T} \delta_{2,j}^{<t>} \cdot h_k^{<t-1>} \tag{30}$$

Derivatives for the weights connected from the input nodes to the hidden state nodes:

$$\frac{\partial}{\partial w_{j,k}^{hx}} J(W) = \sum_{t=1}^{T} \frac{\partial}{\partial z_{2,j}^{<t>}} J(W) \cdot \frac{\partial z_{2,j}^{<t>}}{\partial w_{j,k}^{hx}} \tag{31}$$

$$= \sum_{t=1}^{T} \delta_{2,j}^{<t>} \cdot x_k^{<t>} \tag{32}$$

Now we have the derivatives of $J(W)$ w.r.t. all the parameters in the network for a single training example. This process is repeated for each training example and the derivatives are summed over all training examples. The gradients are then passed to gradient descent to run a single iteration of weight updates. This process is repeated until convergence of the weights or a preset number of epochs.

3.2.4 Gated Recurrent Unit

Natural language can have very long term dependencies in terms of words in a sentence. For example, consider the sentences below:

- The **cat** had a feast lavishing on fish, fruit and milk, **it was** very full.
- The **cats** had a feast lavishing on fish, fruit and milk, **they were** very full.

Notice that a change in only a single word effected the required choice of words later on in the sentence.

A basic RNN will read the sentences above word by word and therefore, the hidden state vector $h^{<2>}$ should effect the hidden states $h^{<14>}$, $h^{<15>}$. But this is very unlikely because the hidden state at each timestep $h^{<t>}$ is completely replaced by a new hidden state calculated in the next timestep $h^{<t+1>}$.

Then there's the problem of exploding and vanishing gradients while training RNN's [4]. Exploding gradients refer to the situation where error terms and therefore derivatives of weights increase exponentially as they are back propagated deeper down the network during back propagation. This will cause gradient descent to diverge away the global minimum. Exploding gradients can be fixed trivially by clipping gradients to a predefined maximum value.

Conversely, vanishing gradients occur when derivatives of weights decrease exponentially towards 0 during back propagation, causing gradient descent to converge very slowly or stop converging all-together. Especially, for long sequences. The gradients are said to "vanish" because values very close to 0 will be rounded to 0, due to numerical round off errors which occur when multiplying floating point numbers with finite precision. In DNN's vanishing gradients applies mostly to layers near the input layer as these will be effected the most. On the other hand, RNN's only have a single hidden layers. However, due to BPTT, error terms and therefore gradients must be back propagated from timesteps $T, ..., 1$. This means the vanishing gradient poses a big problem for basic RNN's especially if the input sequence has many timesteps, which is the case for long sentences.

Gated Recurrent Units (GRU) [7] and Long Short Term Memory (LSTM) [12] are adaptions of the RNN to overcome these issues. Starting with the GRU, intuitively it works by having an update gate Γ_u and a relevance gate Γ_r. A candidate for the hidden state at the current timestep $\tilde{h}^{<t>}$ is calculated which would usually totally replace $h^{<t>}$ in the basic RNN. However, the update gate dictates how much of the candidate $\tilde{h}^{<t>}$ should be incorporated into

the current hidden state $h^{<t>}$ and therefore how much of the previous hidden state $h^{<t-1>}$ should be forgotten. The relevance gate is an indication for how relevant $h^{<t-1>}$ is to the candidate hidden state $\tilde{h}^{<t>}$.

Concretely, a GRU will iterate over the following equations:

$$\Gamma_r = \sigma(w_r[h^{<t-1>}, x^{<t>}] + b_r) \tag{33}$$

$$\Gamma_u = \sigma(w_u[h^{<t-1>}, x^{<t>}] + b_u) \tag{34}$$

$$\tilde{h}^{<t>} = \tanh\left(w_h[\Gamma_r \odot h^{<t-1>}, x^{<t>}] + b_h\right) \tag{35}$$

$$h^{<t>} = \Gamma_u \odot \tilde{h}^{<t>} + (1 - \Gamma_u) \odot h^{<t-1>} \tag{36}$$

Note that $[h^{<t-1>}, x^{<t>}]$ denotes vertical concatenation, $\odot$ denotes element wise multiplication and Γ_r, Γ_u, $\tilde{h}^{<t>}$, $h^{<t>}$ are h_s dimensional row vectors where h_s is the number of nodes in the hidden state layer.

GRU's mitigate the vanishing gradient problem by strategically memorising the same hidden state over many timesteps and replacing it with a new hidden state when it is no longer needed for future predictions. This happens because the sigmoid non-linear activation σ which is used by the update gate Γ_r transforms its inputs to values very close to 0 or very close to 1. Therefore, when memorising the same hidden state, the vector $w_r[h^{<t-1>}, x^{<t>}] + b_r$ will contain large negative numbers making the sigmoid function output values which are approximately 0. This means the same hidden $h^{<t-1>}$ will be retained to the next timestep $h^{<t>}$.

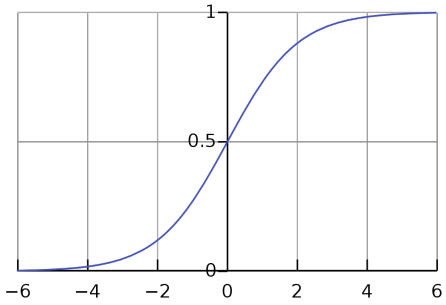

Figure 9: Sigmoid function σ. Notice most of the input values lead to output values very close to 0 or very close to 1.

This memorisation of the same hidden state over many timesteps has the effect of back propagating error terms and therefore derivatives over much fewer timesteps during BPPT, which will greatly alleviate slow convergence caused by vanishing gradients.

3.2.5 Long Short Term Memory

Long Short Term Memory (LSTM) [12] are much the same as GRU's although they did come some 18 years prior. They use three gates instead of two and introduce a new vector called the cell state. This cell state will allow LSTM's to better memorise long term dependencies in input sequences due to similar reasons to how the GRU can memorise long term dependencies. Concretely, the LSTM will iterate over the following equations:

$$\Gamma_u = \sigma(w_u[h^{<t-1>}, x^{<t>}] + b_u) \tag{37}$$

$$\Gamma_f = \sigma(w_f[h^{<t-1>}, x^{<t>}] + b_f) \tag{38}$$

$$\Gamma_o = \sigma(w_o[h^{<t-1>}, x^{<t>}] + b_o) \tag{39}$$

$$\tilde{c}^{<t>} = \tanh\left(w_c[h^{<t-1>}, x^{<t>}] + b_c\right) \tag{40}$$

$$c^{<t>} = \Gamma_u \odot \tilde{c}^{<t>} + \Gamma_f \odot c^{<t-1>} \tag{41}$$

$$h^{<t>} = \Gamma_o \odot c^{<t>} \tag{42}$$

LSTM's and GRU's have comparable performances [8]. However, LSTM's are historically tried and tested. Whereas, GRU's require less computation since they only use two gates instead of three. We will LSTM's, except for very large models where the GRU makes more sense because of computational limitations.

3.2.6 Deep Recurrent Neural Networks

The RNN's described above can be extended to become deep networks [11]. A deep RNN works in a similar manner to a single layer RNN, the only difference is that RNN's are stacked on top of each other within each time step. The output of a RNN will be fed as the input to the RNN in the next layer at each time step. Stacking RNN's makes the model deeper and can be seen as recombining the learned representation from prior layers to create new representations at high levels of abstraction.

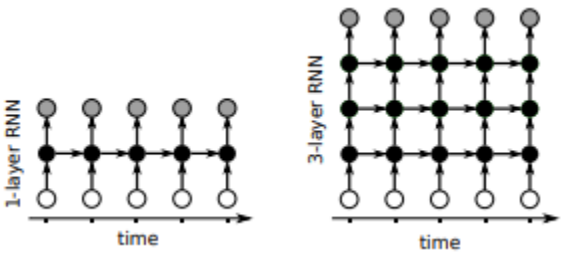

Figure 10: A visualisation of a single layer RNN and a three layer RNN [11].

3.3 Embeddings

3.3.1 Word Embeddings

Word embeddings provide an effective method for represented words as fixed length vectors. The model will have a vocabulary of words and each word could be represented as a one hot vector. The one hot vector will have a dimensionality of V, where V is the vocabulary size. For example, with a vocabulary size of 50,000, a word can be expressed as a one hot vector with dimensionality 50,000, $onehot \in \mathbb{R}^{50,000 \times 1}$. The one hot vector will contain all zero's, except for the index corresponding to the word to encode which will contain a one.

The issue with one hot vector representation is that words with similar meanings will usually have very different vector representations. What this means is that they will be far apart within the vector space. For example, the words "king" and "queen" have index position 1,500 and 30,000 respectively within a vocabulary list. Their one hot vector representations will be markedly different from each other even though they have similar meanings. This will make it increasingly hard for any model to learn correct output sequences.

Word embeddings solve this issue by learning an embedding matrix $E \in \mathbb{R}^{V \times N}$ where N is the chosen length of each embedding usually between 128 and 1024 and V is the vocabulary size. In this way each word in the vocabulary gets a unique word embedding with dimensionality N. The word embedding will group words with similar meanings together so they are close together in the vector space. This grouping can be visualised using the t-SNE algorithm [22].
There are various different models which can learn the embedding matrix E based on a large text corpus using unsupervised methods. Global Vectors for Word Representation (GloVe) [26] will be used to obtain the embedding matrix to be used in our models. This is because there is a publicly available pretrained embedding matrix learned by GloVe on a text corpus of over 1 billion words.

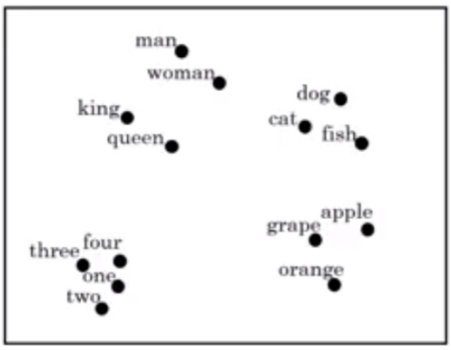

Figure 11: Dimensionality reduction using t-SNE to visualize proximity of similar words within the vector space.

3.3.2 Segment Embeddings

Segments embeddings are useful in allowing models to be aware of different segments within the input sequence. Specifically, some models will take an input sequence which is the persona sentence prepended to the actual input message. A separator token will be placed between the combined persona input message, but this is too weak of a marker.

The RNN models will inherently struggle to distinguish persona from message in the input sequence. To solve this we, will learn a new embedding on top of the word embedding jointly to encode segment tokens $< PSN >$ and $< MSG >$. The word embeddings and segment embeddings will then be summed together and the models will use this as the input $x^{<t>}$.

3.4 Sequence to Sequence Model

Sequence to Sequence learning works by firstly having an encoder RNN which reads the input sequence one token at each time step to produce a fixed length hidden state vector which is then passed to a decoder RNN. Specifically, the last hidden state produced by the encoder is passed as the initial state to the decoder. The decoder will produce an output word at each timestep by taking the prediction at the previous timestep as an input until the end of sequence token $< EOS >$ is predicted. During training, we use teacher forcing which works by feeding the ground truth $y^{<t>}$ instead of the prediction $\hat{y}^{<t>}$ as the input to the next time step $x^{<t+1>}$. Using this model we are able to create a many-to-many RNN, where the number of output tokens in a sequence has no direct link to the number of input tokens.

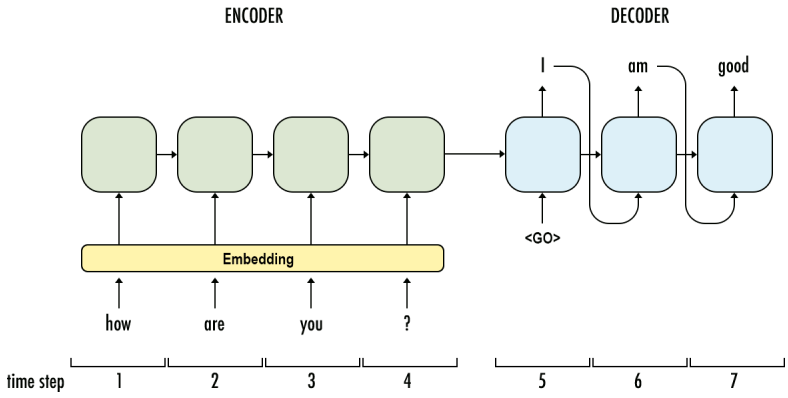

Figure 12: Example of a Sequence to Sequence model choosing a reply to an incoming message.

Within the decoder a softmax activation function over all words in the vocabulary is performed to choose the word with highest likelihood of being the next word in the output sequence.

$$\hat{y}^{<t>} = \underset{softmax(w_y h^{<t>} + b_y)}{\arg\max} \tag{43}$$

3.5 Attention

Attention is one of the most influential and powerful ideas in deep learning to recent date. It was originally introduced in 2014 by Bahdanau et al [3] to improve Neural Machine Translation. However, the concepts described in this paper can be applied to any Sequence to Sequence model. The problem with vanilla Sequence to Sequence models is that the encoder is forced to encode the meaning or context of an input sentence into a fixed length vector which is passed to the decoder. Therefore, it will struggle to deal with long sentences. Especially, sentences longer than those found in the training set because the encoder will find it very challenging to include the whole context of the input sentence into a fixed length vector, some information in the input sentence will be lost.

Attention solves this issue by allowing the decoder, at each time step, to choose a set of words within the input sequence to put its "attention" on. Therefore, the decoder can get information about the input sequence not only from a fixed length vector produced by the encoder, but also the relevant context within the input sequence during each decoder timestep.

3.5.1 Encoder

Firstly, lets look at how the Attention Model modifies the encoder in a standard Sequence to Sequence Model. A Bi-directional Recurrent Neural Network (BRNN) is used within the encoder. BRNN's work by forward propagating the input sequence through the network as usual, but once the input sequence has been fully read, it will be read one token at a time the reversed input sequence to produce another set of hidden state vectors. Now each token within the input sequence is associated with two separate hidden state vectors instead of only one. They can be concatenated together to make one fixed length vector. By doing this, at each time step the encoder is not only capturing information about previous words but also following words within the current context.

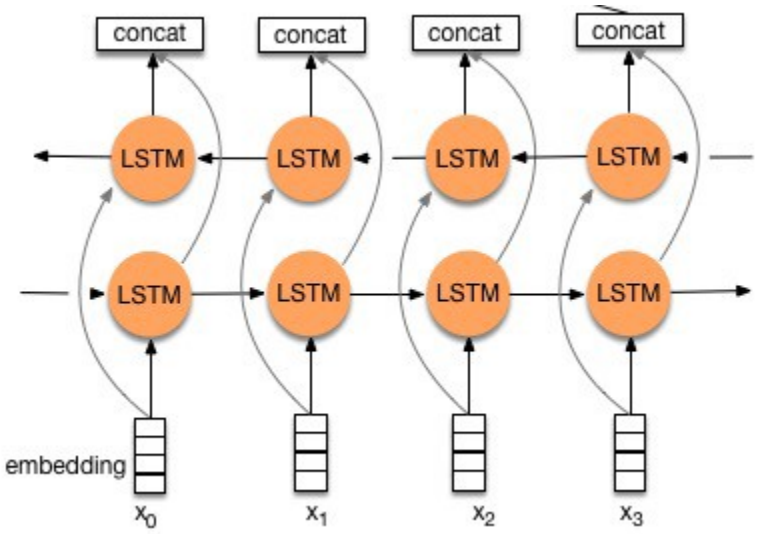

Figure 13: Illustration of the BRNN. It has a forward layer and backwards layer. The forward layer is computed through forward propagation of the input sequence. The reverse layer is computed through forward propagation of the reversed input sequence.

The concatenated hidden state vector at each time step t within the encoder and the hidden state vector for the decoder will be denoted as $\bar{h}^{<t>}$ and $s^{<t>}$ respectively. The context vector at time step t within the decoder, not to be confused with the cell state in LSTM's, will be be denoted as $c^{<t>}$. Even though we are doing bi-directional encoding, this is just for the purpose of attention, the final forward encoded hidden state $\overrightarrow{h^{<T_x>}}$ is used as the decoder initial state $s^{<0>}$.

3.5.2 Decoder

The decoder at timestep t will will produce output $\hat{y}^{<t>}$ which will be the highest probability from:

$$P(\hat{y}^{<t>}|\hat{y}^{<1>}, ..., \hat{y}^{<t-1>}, X) = g(\hat{y}^{<t-1>}, s^{<t>}, c^{<t>}) \tag{44}$$

$$s^{<t>} = f(\hat{y}^{<t-1>}, s^{<t-1>}, c^{<t>}) \tag{45}$$

Notice the addition of a context vector to the decoder. This context vector aims to capture the most relevant hidden state vectors within the encoder. The context vector at each timestep is calculated by performing a weighted sum over all hidden state vectors that were obtained from the encoder:

$$c^{<t>} = \sum_{j=1}^{Tx} \alpha^{<t,j>} \bar{h}^{<j>} \tag{46}$$

each weight $\alpha^{<t,j>}$ represents how relevant as a score between zero and one the input word j is to the current output at timestep t. The weights over all hidden state vectors in the encoder for the current decoder timestep is computed by a softmax function:

$$\alpha^{<t,j>} = \frac{exp(e^{<t,j>})}{\sum_{i=1}^{Tx} exp(e^{<t,i>})} \tag{47}$$

$e^{<t,j>}$ describes the amount of attention $\hat{y}^{<t>}$ should pay to the encoder hidden state vector $\bar{h}^{<j>}$. This function of attention is not known before hand as it varies between data sets and tasks but it is learned by a small feed-forward Artificial Neural Network also called an alignment model, usually with only one hidden layer. Parameters for this network are learnt jointly with the Attention Model by back propagating gradients.

$$e^{<t,j>} = a(s^{<t-1>}, \bar{h}^{<j>}) \tag{48}$$

The impact of Attention on BLEU scores for Neural Machine Translation can be seen in the figure below. [25].

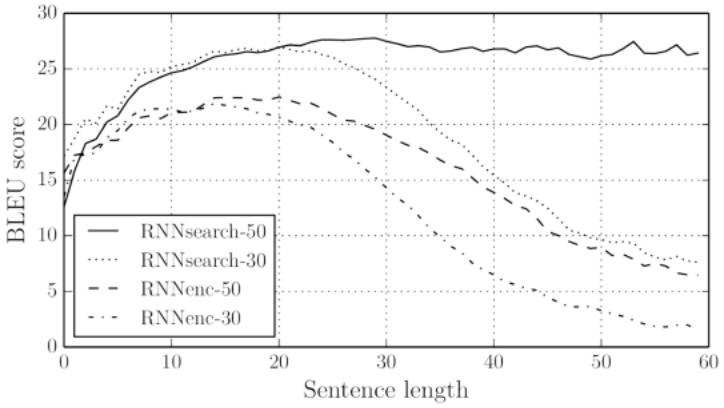

Figure 14: RNNsearch-50 was trained with Attention and maintains a consistent BLEU score even as sentence length increases [3].

The above describes a general framework for attention. Concretely, in our models we follow Luong's modified approach to effective additive attention [21]. The context vector $c^{<t-1>}$ will be concatenated to the input $x^{<t>}$ for all output timesteps $1, ..., T_y$. Also, the alignment model uses $s^{<t>}$ instead of $s^{<t-1>}$, to perform the computation below:

$$e^{<t,j>} = v^{\top} \tanh(w_a s^{<t>} + w_b \bar{h}^{<j>}) \tag{49}$$

Where $w_a \in \mathbb{R}^{a_{dim} \times h_s}$, $w_b \in \mathbb{R}^{a_{dim} \times 2 \cdot h_s}$, $v \in \mathbb{R}^{h_s}$ and a_{dim} is the attention dimensionality hyper-parameter. On top of this, the context vector calculated at the current timestep $c^{<t>}$ will be concatenated to $s^{<t>}$ to form the prediction $\hat{y}^{<t>}$ as follows:

$$\hat{y}^{<t>} = \underset{softmax(w_y[s^{<t>}, c^{<t>}] + b_y)}{\arg\max} \tag{50}$$

3.6 A New Model

In this project, we experiment with a novel model specifically designed to handle two different input sequences simultaneously, that is the persona input which is simply a series of facts about the chatbot and an input message from the user. For example:

- Chatbot Persona: "I am 19 years old. I am a student. I live in England. I like football"
- Message: "Hi, how are you doing today?"

To produce a single output series, for example "I'm fine, and you?".

3.6.1 Encoders

This model will have two separate bi-directional GRU encoders, each of which is performing the same operations as a standard encoder but on different input sequences. In our case there will be an encoder responsible for reading the persona and another encoder responsible for reading the message.

The final hidden states for the persona encoder and message encoder are concatenated and then reduced to the hidden state size of the decoder. This is what the decoder will use as it's initial state.

$$s^{<0>} = \tanh\left(w_{s0}[\overrightarrow{h^{T_p}}, \overrightarrow{h^{T_m}}] + b_{s0}\right) \tag{51}$$

Where $\overrightarrow{h^{T_p}}$ and $\overrightarrow{h^{T_m}}$ are the final forward hidden states of the persona and message encoder respectively. This will keep the dimensionality of the hidden states for the two encoders and the decoder the same. Furthermore, as the weight matrix and bias w_{s0}, b_{s0} will be trained as part of the model, the model can learn how to forge persona and message representations into a combined representation which the decoder can use as it's initial state meaningfully.

3.6.2 Decoder

The attentive decoder is much the same, except now it can pay attention to two separate encoder hidden state sequences. This has the advantage of explicitly separating the persona sequence from the message sequence. Consequently, allowing the decoder to pay attention to any word in the message and any word in the persona simultaneously.

Firstly, there will be separate alignment models for the persona encoder and message encoder which calculate the two context vectors $c_p^{<t>}$ and $c_m^{<t>}$. This is done using the current decoder hidden state $s^{<t>}$ as discussed in the previous section. The input $x^{<t>}$ at each decoder timestep becomes $[x^{<t>}, c_p^{<t-1>}, c_m^{<t-1>}]$. The output at the decoder $\hat{y}^{<t>}$ is calculated by:

$$\hat{y}^{<t>} = \underset{softmax(w_y[s^{<t>}, c_p^{<t>}, c_m^{<t>}]+b_y)}{\arg\max} \tag{52}$$

This model has an extra encoder and therefore a considerably higher number of parameters, especially when using deep RNN's. For this reason, this model will strictly use GRU's as they requires less computation then LSTM's, making them a more scalable choice. Furthermore, GRU's do not have an associated final cell state to pass from encoder to decoder along with the final hidden state, simplifying the model somewhat.

3.7 Transformers

Transformers [39] are a revolutionary invention in the Natural Language Processing field. Currently, state-of-the-art Neural Machine Translation models are some from of Transformer. This makes them an extremely worthy model to include in this project because in many ways the problem we are trying to solve is similar to Neural Machine Translation.

Historically, LSTM's and GRU's have dominated the Natural Language Processing field, but they have some problems:

- They still struggle to learn many long term dependencies found in natural language because the cell state $c^{<t>}$ in LSTM's or the hidden state $h^{<t>}$ in GRU's must pass through a linear number of timesteps between any two related input tokens.
- Outputs $\hat{y}^{<t>}$ are calculated temporally which doesn't make full use of highly parallelisable hardware such as GPU's.

Transformers don't suffer from these drawbacks. This is because they aim to bridge this gap between related words in the input by doing away with the idea of timesteps all-together. The is done by allowing any word in the input sentence to attend to any other word in the input sentence. Transformers can do this by relying solely on Dot-Product attention [21]. Interestingly, Transformers seem to have a better understanding of language, as they perform much better on Winograd schemas when compared to their LSTM or GRU counterparts [14].

Concretely, they have an encoder and a decoder. The encoder performs Self Attention on the input sentence and the decoder does the same with the output sentence generated so far. With each forward pass of the Transformer, it produces an extra word which is appended to the output sentence generated so far, until the $< EOS >$ token is predicted.

Firstly, an input matrix X is created by populating it with the relevant indicies of the embedding matrix for the words in the sentence. The Transformer inherently has no recurrence and therefore similar to ANNs, has no knowledge about the order of words in the input sentence. To solve this the Transformer adds a positional encoding to each word embedding vector in X to bake into their representations a sense of their relative position in a larger sentence. The positional encoding are represented using trigonometric functions at different frequencies as follows:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{53}$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{54}$$

Where pos is the position of the word being encoded. Each word is a d_{model} dimensional vector which is equivalent to the embedding dimension. $i \in [0, ..., \frac{d_{model}}{2}]$ is the dimension. In simple terms, this is producing a vector which is the same size as the word embedding vector. At even indicies of this vector the Sine function is used and at odd indicies of the vector the Cosine function is used. These functions will form a geometric progression from 2π to $10000 \cdot 2\pi$ in the wavelengths as i increases across the dimensions from 0 to $\frac{d_{model}}{2}$. Once we have the model input matrix X for the current sentence we can move on to Self Attention.

Self Attention works by creating Query, Key and Value matrices from the whole input sentence matrix X as follows:

$$Q = Xw_Q \tag{55}$$

$$K = Xw_K \tag{56}$$

$$V = Xw_V \tag{57}$$

Where $X \in \mathbb{R}^{m \times d_{model}}$ and m is the number of words in the input sentence. $w_Q \in \mathbb{R}^{d_{model} \times d_k}$, $w_K \in \mathbb{R}^{d_{model} \times d_k}$, $w_V \in \mathbb{R}^{d_{model} \times d_v}$. d_k is usually chosen to be much smaller then d_{model} to make Self Attention less computationally expensive. We then perform Self Attention on the calculated Queries, Keys and Values as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^\top}{\sqrt{d_k}})V \tag{58}$$

The equation above can be thought of performing Dot-Product attention for each word on each word as a vectorised implementation. The Dot-Product can be thought of as a measurement of similarity between two vectors. Therefore, $QK^\top \in \mathbb{R}^{m \times m}$ is measuring the similarity between each query individually to all the keys. Multiplying element-wise by $\frac{1}{\sqrt{d_k}}$ results in more stable gradients. This is because for large values of depth, the dot product becomes large in magnitude, which in turn means the softmax function will have minuscule gradients, resulting in a hard softmax, but this is mitigated by the division. A softmax over each row in the resulting matrix is performed. This will normalise all Dot-Products for each query to make them sum to 1.0, so they can treated as scores. Finally, the resulting matrix is multiplied by the Values matrix V. Intuitively, this is a vectorised implementation of a weighted sum over all individual values where the corresponding weights are within each row of the $softmax$ matrix.

Self Attention is applied to different sets of Query, Key and Value matricies calculated from the same input matrix X. This is called Multi-Head Self Attention. Having multiple heads expands the models ability to focus on different positions and it gives the Self Attention layer multiple representation subspaces.

$$MultiHead(X) = Concat(head_1, ..., head_h)W^O \tag{59}$$

$$head_i = Attention(XW_i^Q, XW_i^K, XW_i^V) \tag{60}$$

Where h is the number of heads and the learnt weight matricies are of dimensions $W^O \in \mathbb{R}^{h \cdot d_v \times d_{model}}$, $w_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $w_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $w_i^V \in \mathbb{R}^{d_{model} \times d_v}$, $head_i \in \mathbb{R}^{m \times d_v}$ and the concatenation is performed horizontally. The encoder stacks N layers of Multi-Head Self Attention together. This is done by feeding the output of a Multi-head Self Attention layer through a small Feed Forward MLP, followed by a Layer Normalization layer [2] with a residual connection to the Multi-Head Self Attention output. The residual connections mitigate the vanishing gradient problem. All together, that makes one layer, the resulting matrix is then passed as the input to the next layer of Multi-Head Self Attention. The decoder side utilises Multi-Head Self Attention and Decoder-Encoder Attention. Decoder-Encoder Attention is very similar to Multi-Head Self Attention, except it will use the Query matrix from the Multi-Head Self Attention layer below and Key, Value matricies will be the encoder output. After passing through all layers of the encoder and decoder a single word can be predicted using a $softmax$ over the whole vocabulary.

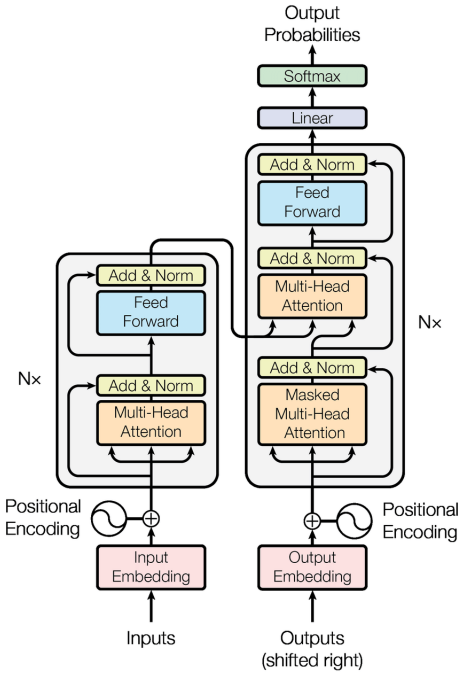

Figure 15: The Transformer. It has N layers of Multi-Head Self Attention. The left half belongs to the encoder and right half to the decoder.

Lastly, the efficient Adam optimizer with a custom learning rate scheduler is used to train the Transformer. The learning rate is updated according to the following formula:

$$\alpha = d_{model}^{-\frac{1}{2}} \cdot min(step\_num^{-\frac{1}{2}}, step\_num \cdot warmup\_steps^{-\frac{3}{2}}) \tag{61}$$

$warmup\_steps$ value of 4000 are used in these experiments.

3.8 Beam Search

Beam Search is a heuristic search algorithm which has been shown to outperform Greedy Search for generating responses to an input [36]. The decoders in the models we have described above are calibrated to find the probability of the current word over the vocabulary, given the words in the input sentence $x^{<1>}, ..., x^{<T_x>}$. The words in the input sentence are represented by a series of vectors v, which may contain the final hidden state of the encoder, context vectors from attention and so forth, depending on which model is used. Therefore, the decoder is trained to estimate the following probability:

$$P(\hat{y}^{<1>}, ..., \hat{y}^{<T_y>} | x^{<1>}, ..., x^{T_x}) = \prod_{t=1}^{T_y} P(\hat{y}^{<t>} | v, \hat{y}^{<1>}, ..., \hat{y}^{<t-1>}) \tag{62}$$

Greedy Search is exactly that, greedy. At each decoder timestep t, the decoder will pick the word $\hat{y}^{<t>}$ which has the highest probability. The problem with this approach is that the word with the highest probability $\hat{y}^{<t>}$ may be the best choice in the short term, however choosing this word will effect all future prediction for words in the current time series $\hat{y}^{<t+1>}, ..., \hat{y}^{<T_y>}$. Therefore, it's possible choosing another word at $\hat{y}^{<t>}$ which doesn't have the highest probability, will overall effect the future predictions in such a way that leads to a larger overall probability of the output sequence $P(\hat{y}^{<1>}, ..., \hat{y}^{<T_y>}|x^{<1>}, ..., x^{T_x})$, according to the model.

Searching through all possible combinations of words to find a sentence which maximizes the probability $P(\hat{y}^{<1>}, ..., \hat{y}^{<T_y>} |x^{<1>}, ..., x^{T_x})$ given by the model, is clearly computationally unrealistic. Beam Search is a better approach.

It works by maintaining B different beams. A beam can be considered to be a current possible output sequence $y^{<1>}, ..., y^{<t>}$ which is different to the other beams. At each timestep we will calculate a cumulative log likelihood probability of the B most likely words for each beam, apart from the first timestep where each beam will start with a different word from the B most likely first words to ensure all the beams start differently. This temporarily gives B^2 possible output sequences when $t \neq 1$, by considering $B \times V$ different probability outputs from the decoder. From the B^2 possible output sequences, we choose the most likely B output sequences using the length normalized summed cumulative log likelihood probability we calculated previously. The summed cumulative log likelihood is length normalized by dividing it by the length of words currently in the beams. Doing this will prevent beam search from incorrectly favouring short sentences over long ones. This process is repeated until all beams B end in the $< EOS >$ token.

3.9 Evaluation Metrics

Both automatic metrics and human evaluation will be used to quantify the effectiveness of the chatbot. The following automatic metrics will be used:

1. **Perplexity:** This is a measurement of how well the model predicts test data. Intuitively, perplexity means the inability to understand something complicated or unaccountable, so the lower the perplexity, the better. It is calculated by performing the log likelihood of the correct sentence over the whole test set given by: $\frac{1}{m} \sum_{i=1}^{m} \sum_{t=1}^{T_y} \log \left(P(y^{<t>})\right)$.

2. **F1:** This measures world overlap by calculating: $2 \cdot \frac{precision \cdot recall}{precision + recall}$ where *precision* is the fraction of words in the generated response that are also in the ground truth response and *recall* is the fraction of words in the ground truth response, that are in the generated response.

BLEU [25] is a useful metric for Machine Translation but not for this problem. This is because BLEU score is based solely on the number of matching n-grams between the output sentence from the model and the ground truth output from a hidden test set. Naturally, this can work for translations since there is only a limited set of correct translations to an input sentence. However, for our problem there is a much larger set of possible replies to an input message, where two very different replies could be equally valid. All the automatic metrics somewhat fall victim to this phenomenon [37], but especially BLEU [20]. For this reason, BLEU won't be used. Conversely, human evaluation metrics are much better predictors for the quality of replies produced by a model. A human will converse with the chatbot for a total of 12 utterances. After this the human will be asked to rate the chat-bot based on the following metrics:

1. **Fluency:** This measures between 0-5, how much the chatbot answers make sense given the context of the conversation.

2. **Engagingness:** This measures between 0-5 how engaging the chatbot is during the conversation.

3. **Consistency:** This measures between 0-5, how consistent the chatbot is with its answers during the conversation.

4. **Persona Detection:** At the end of the conversation the human will be given a random persona within from the list of persona's and the persona which the chat-bot was using for the conversation. They will then have to predict which persona the chatbot was using.

When obtaining the human evaluation metrics, humans will not be aware if they are talking to a chatbot or another real human. By doing this, human evaluation metrics can be collected for a real life human. This is useful because

we can compare these to the metrics of the chatbot while removing any bias. Overall it's difficult for a chat-bot to achieve good scores for all these metrics simultaneously, however the main goal of this project it to try to maximize persona detection and consistency as much as possible.

4 Design

In the previous section, we have understood many theoretical concepts about how the models employed in this project are configured and trained.

In this section will look at some concrete configurations of the models described above in terms of hyper-parameters and datesets which they may use, with guidance from the relevant literature. On top of this, we specify the design for a command line interface used as a learning framework to train, evaluate and interact with the models described above. Furthermore, we show the design for a website used to deploy the models so that users can interact with them easily.

4.1 Hyper-Parameters

Hyperparameters are the model configurations which cannot be learnt through back propagation unlike intrinsic parameters, for example the number of layers in an ANN. Commonly, hyperparameters are learnt through either a static train-valid-test split of the dataset or K-Fold Cross-Validation. Using a static train-valid-test split, where the model is trained only a few times on hyperparameters which are suspected to be optimal is extremely difficult to do within the limited time frame of this project. This is because training a model only once can take up to 10 days, even while using computationally powerful hardware.

For this reason we will follow the recommendations of Britz et al [5]. They perform a massive exploration of RNN architectures to find the general optimal architecture for solving Neural Machine translation, which is in many ways analogous to our own problem. Using their conclusions we will outline the hyper-parameters for each model, starting with the Sequence to Sequence with Attention model:

- RNN Type: LSTM
- Hidden State Dimensionality: 512
- Embedding Dimensionality: 300
- Attention Type: Additive
- Attention Dimensionality: 512
- Encoder Depth: 4
- Decoder Depth 4

Next is the new model which we shall call Multiple Encoders:

- RNN Type: GRU
- Hidden State Dimensionality: 512
- Embedding Dimensionality: 300
- Attention Type: Additive
- Attention Dimensionality: 512
- Encoder Depth: 4
- Decoder Depth 4

Pope et al [27] have performed a similar experiment to find the optimal Transformer architecture:

- Hidden dimensionality d_{model}: 1024

- Multi-Head Self Attention heads h: 16
- Multi-Head Self Attention layers N: 6
- Query, Key Dimensionality d_k: 128
- Value Dimensionality d_v: 128
- Feed Forward MLP: 3 layers, input layer has d_{model} nodes. Single hidden layer has 4096 nodes with $ReLu$ activation function. Output layer has d_{model} nodes with a linear activation function.
- Residual Dropout: 0.3
- Maximum Training Sentence Length: 99^{th} percentile of all sentence lengths.

All models will use a beam width of 10 with a beam penalty of 1.0. They will also share the same embedding between Encoders and Decoders, as this can reduce the number of trainable parameter by up to a third, while have little effect on performance [28]. Regarding Batch size, "It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize" [13]. For RNN models in this project, large batch sizes to the extent of which the previous quote is referencing is not achievable because each training example must have a one-hot label, which is a $T_y \times V$ dimensional matrix. Therefore, each training example will consume a relatively large amount of memory, making large batch sizes impossible because of Out Of Memory (OOM) errors. For this reason, RNN based models will use the maximum possible batch size which is excepted to be 64 while using an 11GB GPU. Conversely, experiments on Transformers have shown a larger batch size is always better [27], so the same rule will apply here.

It's not clear what the number of training epochs should be for each model, as our task is unique and hasn't been tried and tested by researchers. Consequently, we will use Early Stopping [6]. This works by calculating the the loss on the whole validation set, by using the Categorical Cross Entropy cost function at the end of each epoch. If the model achieves a validation loss which is less then any other validation loss seen so far in the current training cycle, then the model will be saved to file. The model will continue to train until a predefined number of epochs have passed without saving the model to file. By doing this the model which performs best on the validation set and therefore the model with the highest generalisation ability will be saved, preventing overfitting and the need to set a predfined number of epochs.

4.2 Datasets

Pretraining Sequence to Sequence models has been shown to increase BLEU scores by over 13% for Neural Machine Translation tasks [29]. Generally, for a model to perform well in Neural Machine Translation it must firstly learn the semantics and syntax of language and then learn how to generate correct translations. The same applies to our problem. Expecting a model to become fluent in language and generate meaningful replies based on a persona with limited training is unrealistic, especially if the persona based dialogue dataset is not huge. Therefore, we will pretrain all models on two datasets not related to persona first, so that they can build an initial understanding of the English language.

Firstly, models will be pre-trained on the Cornell Movie Dialog Corpus [9]. This dataset contains 220,579 conversational exchanges between 10,292 pairs of movie characters with a total of 304,713 utterances. Thereafter, the models will be pre-trained on the Daily Dialog Dataset [18]. Daily Dialog contains 13,118 conversations with a total of 103,632 utterances. It is also a higher quality dataset when compared to the Cornell Movie Dialog Corpus because Daily Dialog was manually crafted for the problem of generating high quality responses within everyday conversations, but it lacks any persona data.

Succeeding pretraining, comes training the models on the PERSONA-CHAT dataset [43]. This dataset includes a large corpus of conversations between two individuals. Each individual is given a persona as a set of 5 sentences describing background facts about themselves. The individuals are then given the task to get to know each other. This resulted in over 140,000 utterances, 19,000 conversations and 1,355 different personas.

	# examples	# dialogues	# personas
Training Set	131,438	17,878	1,155
Validation Set	7,801	1,000	100
Test Set	6,634	1,015	100

Table 1: Number of utterances, conversations and personas collected in the data set.

4.3 Data Preprocessing

Data for each of these datasets take the form of raw text. Machine learning models require matricies of numbers which represent the cleaned raw text. Since this is a supervised learning problem, they also require labels for each training example. Here, we will establish a general strategy to do this.

Firstly, for each dataset the raw text must be organised into a list of triplets, where the first element in each triplet is the persona, the second is the message and the third element is the reply. If the dataset doesn't contain any persona data, then the persona element will simply be an empty string. Secondly, each element in each triplet needs to cleaned into a form which will make learning for the models easier. Cleaning will be done by:

- Removing all non-printable characters.
- Removing all punctuation, except full stops, commas and question marks.
- Replacing all contractions. For example, "it's" becomes "it is".
- Make all characters lowercase
- Space punctuation for tokenization. For example ,"Hi, how are you?" becomes "Hi , how are you ?"

This can be done efficiently by using Regular Expressions. Now we will have a list of cleaned triplets. Next, we need to tokenize each element in each triplet. This means assigning each word a unique index from a fixed vocabulary and then converting raw text to a list of numbers, so we will have a list of triplets, where each element in the triplet is itself a list of numbers. To do this, we first build a vocabulary of words by reading all words from all datasets and assigning each word which appears at-least twice a unique index. The relation between each word and it's unique corresponding index can be stored in a dictionary data structure.

Many machine learning frameworks require time series inputs to all have the same number of timesteps. Therefore, the persona list, message list and reply list within each triplet will be padded with zeros to their maximum lengths across all triplets respectively. Within the models masking of zero values should be used so that this padding does not effect any predictions. The message lists will then be converted to matricies by one-hot encoding each number in each message list over the vocabulary. This is required because these message matricies will be used as the labels in the models. An important note is that this one-hot encoding step should occur batch by batch to avoid OOM errors.

Each model will use each triplet as a single training example. Sequence to Sequence with Attention, prepends the persona list to the message list and this will be the input to the word embedding layer. Trivially, a segment list is created before feeding inputs to the model by creating a persona segment list which contains the persona token $< PSN >$ for all elements in the persona list and another message segment list which contains the message token $< MSG >$ for all elements in the message list. The persona segment list is prepended to the word segment list and this will be fed into the segment embedding layer of the model. Similarly, Transformers will do the same, except they tokenize words using subword tokenization [35]. On the other hand, the Multiple Encoders model explicitly takes care of the persona so there is no need for a segment embedding.

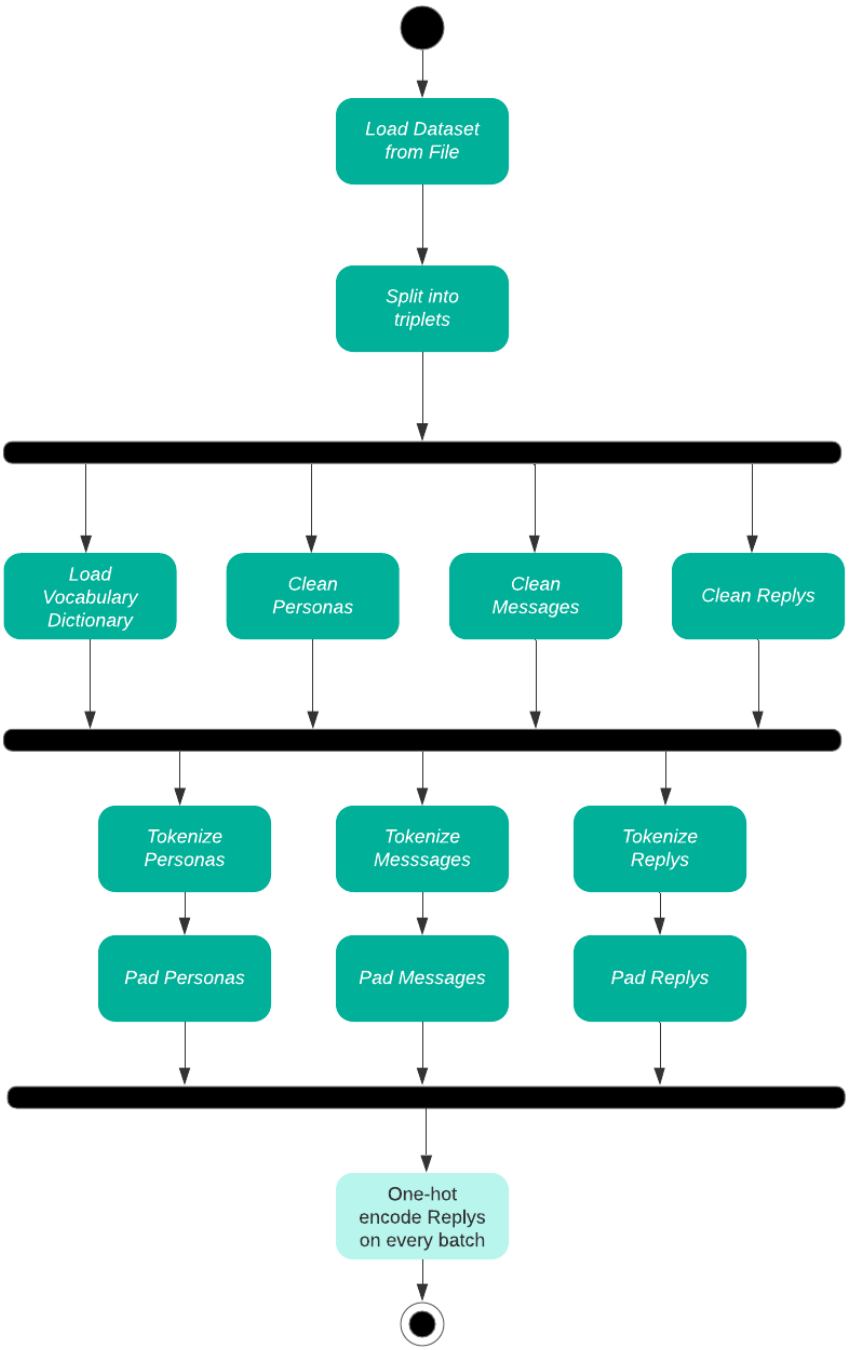

Figure 16: Activity Diagram to show pre-processing workflow.

4.4 Command Line Interface

Alongside this project, we aim to create an easy to use text generation framework to allow users to train, evaluate and conversate with trained models. This will take the form of a Command Line Interface (CLI). Below is a table showing information about the list of arguments the intended CLI can take:

Train Arguments

Arg Name	Type	Description	Default
train	String	Name of the model to train	None
glove_filename	String	Name of GloVe file	glove.6B.300d.txt
batch_size	Int	Training batch size	64
epochs	Int	Maximum number of training epochs	100
early_stopping_partience	Int	Early Stopping Patience	15
verbose	Int	= 1 to print batch loss during training	0
segment_embedding	Bool	Use segment embedding?	True

Evaluation Arguments

eval	String	Name of the trained model to evaluate	None

Talk Arguments

talk	String	Name of the trained model to talk to	None
beam_width	String	Beam length to use during beam search	4
beam_search	Bool	Use beam search?	False
plot_attention	Bool	Plot attention weights	False

Models which can be used will include $seq2seq$, $deep\_seq2seq$, $multiple\_encoders$, $deep\_multiple\_encoders$, $transformer$. Segment embeddings will not be applied to $multiple\_encoders$ and $deep\_multiple\_encoders$ models no matter what value is given for the argument. As an example, to train a Transformer the following command could be used: $python\ main.py\ –train\ transformer\ –batch\_size$ 256. After training has finished, the following command could be used to talk to the trained Transformer: $python\ main.py\ –talk\ transformer$.

4.5 Website

The model which achieves the best results will be deployed to a website to allow users to interact with the chatbot without the need to install any extra software. The front-end should be simple and intuitive with a minimal number of features. Straightforwardly, it should allow the user to enter a message through typing in which case the reply of the chatbot is displayed.

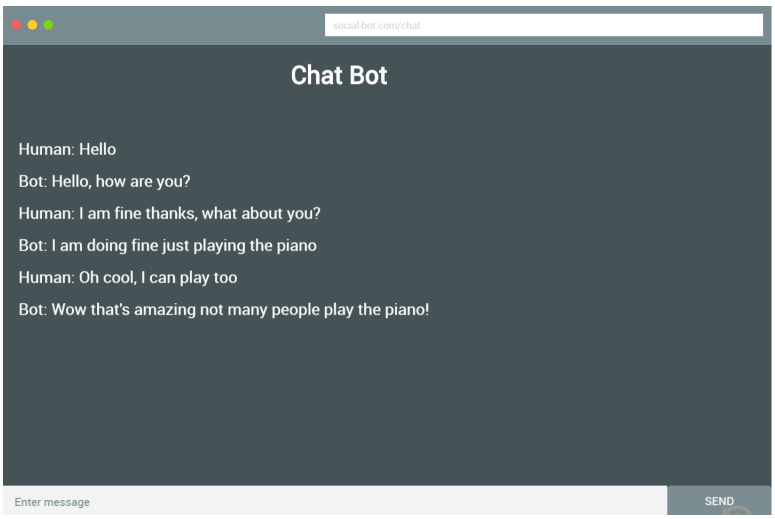

Figure 17: Simple wire-frame for the website GUI.

The back-end will hold the deployed model, it needs to be carefully designed to ensure latency of replies is minimised. Tensorflow Serving is a highly optimised server designed specifically for deploying large scale machine learning models and it is used internally at Google. On the server side we will have a Tensorflow Serving which provides a REST API on port 8501 for clients. Clients can send a HTTP POST containing the input data and get back the inference of the model. Tensorflow Serving has the following benefits:

- **Highly Optimised:** Optimised for making inferences, minimising space and time complexity for inference.
- **Consistent API:** Tensorflow Serving API is well documented making on boarding new members to work on the project easy and increasing maintainability.
- **Batch Inference:** If multiple inference requests are made in a short time period Tensorflow Serving can automatically perform batch inference which results in more efficient computation.
- **Model Versioning:** Different models can be loaded seamlessly simply by uploading the saved model and modifying a configuration file, removing the need for a server reboot.

However, Tensorflow Serving on it's own is not enough. Web clients will have no knowledge of the vocabulary dictionary used during training so there is no way for them to preprocess text data into matrix form required by the models. Therefore, Tensorflow Serving will be hidden behind a Flask server. Clients will send raw text in HTTP POST requests to the Flask server, in which case the Flask server will perform the necessary preprocessing, send this to the Tensorflow Serving and then reply back to the original client with the raw text reply. By doing this we still maintain all the benefits of using Tensorflow Serving, while keeping latency low since the Flask server and Tensorflow Serving will be on the same physical machine.

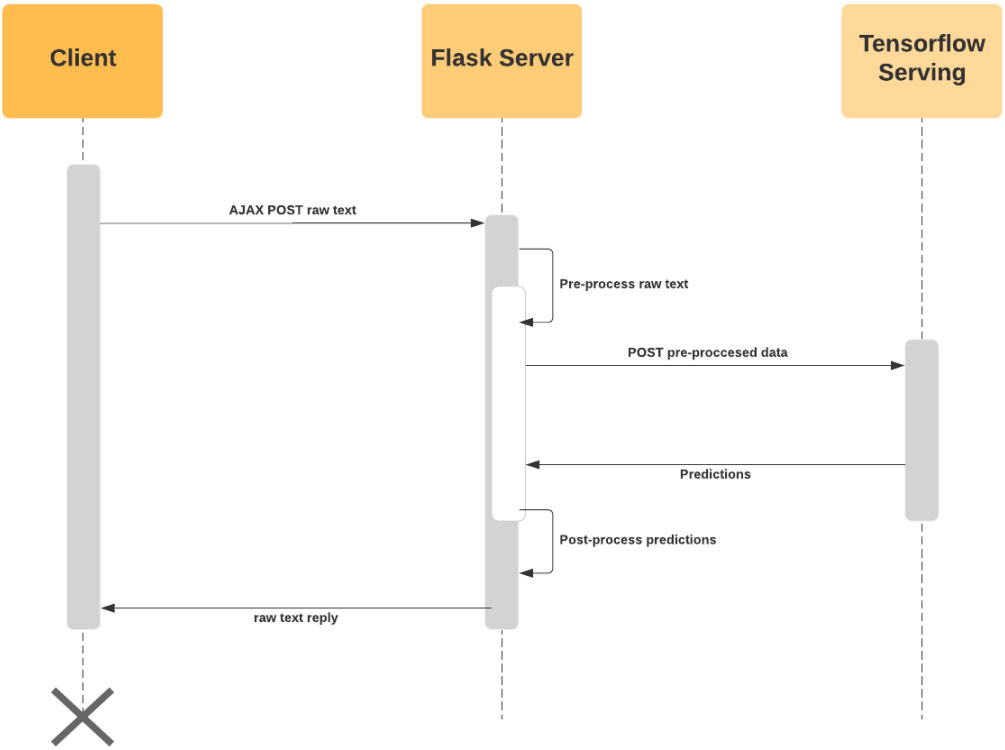

Figure 18: Sequence Diagram for the proposed web architecture.

4.6 Functional Requirements

The following functional requirements specify what the project must do in order to be measured as successful.

1. **Seq2Seq with Attention:** The project will implement deep and shallow versions of the Sequence to Sequence model with Attention, capable of dialogue generation.
2. **Multiple Encoders:** The project will implement a new model called Multiple Encoders, capable of dialogue generation.
3. **Transformer**: The project will implement the Transformer model, capable of dialogue generation.
4. **Comparison of Models**: The models listed above will be trained and compared according to automatic and human evaluation metrics.
5. **Command Line Interface** A CLI will be implemented, to the specification outlined above, which allows users to train, evaluate and conversate with the models above, by specifying their own arguments and hyper-parameters.
6. **Website:** The website will allow the user to enter a message to the chatbot in a text field in which case the chatbot will send back a meaningful reply. The conversation history for the current dialogue and the persona which the chatbot is using will also be shown for the duration of the interaction.

4.7 Non-functional Requirements

Non-functional requirements specify performance and control metrics which need to be met in order for the project to be measured as successful.

1. **Reply Speed:** The user should not have to wait more than 5 seconds for the chatbot to generate a reply. A waiting time longer than this will make the conversation unnatural and cause users to become impatient.
2. **Website Traffic:** The website should be able to handle 5 concurrent users interacting with the chatbot.
3. **Availability:** The website should have an up time of at least 99%.
4. **Reliability:** The website should not crash. If errors do occur they should be dealt with gracefully.
5. **Mobile Friendly:** The website will be built with a mobile first approach to ensure the website is compatible with a range of devices from mobile phones to tablets to large screen monitors.

5 Implementation

In the previous section we discussed the requirements, design and specification for this project. However, implementation details such as the languages to use were not included. Consequently, in this section we specify those details and also outline difficulties faced during implementation.

5.1 Language and Learning Framework

Python is the language chosen for the implementation of this project for the following reasons:

- **Simple and Flexible:** Python is a high level language which abstracts away many complex processes such as memory management, allowing the programmer to focus solely on building efficient machine learning models.
- **Diverse Ecosystem:** Python has a large set of deep learning libraries and frameworks such as Tensorflow, Pytorch and Keras. Python is the go-to language for machine learning applications and is used by technology conglomerates globally.
- **Platform Independent:** Python is supported by many platforms, including Windows, Linux, and macOS.

There are several deep learning libraries which could be used for this project. For this implementation, we chose Tensorflow [24] for the following reasons:

- **High-level API and Flexibility:** Tensorflow provides the best of both worlds, with the ability to quickly create generic models with it's high-level API or to create custom models with the ability to subclass Layers and Models.
- **Efficient Deployment:** Tensorflow Serving provides a standard and optimised way to deploy machine learning models.

Moreover, the website will use HTML, CSS and JavaScript for the client side and Python, Flask and Tensorflow Serving on the server side. Git was used for version control. The public repository can be accessed with the following link: https://github.com/psyfb2/Chatbot

5.2 Training

All models were trained using one of two machines:

- Linux Remote Machine with 4 CPUs, 24GB RAM, 1 GPU (11GB NVIDIA GeForce RTX 2080 Ti)
- Google Cloud Deep Learning VM with 2 CPUs, 16GB RAM, 1 GPU (12GB NVIDIA TESLA K80)

Unfortunately, some hyper-parameter and model architecture tuning was still required as the models didn't perform as well as hoped initially. This did consume a lot of time since it wasn't clear exactly what should be changed. The Sequence to Sequence and Multiple Encoders models seemed to be overfitting the training data because after only a few epochs the validation loss would no longer decrease, meanwhile the training loss did decrease. Generally, overfitting can be solved by decreasing the model complexity or introducing regularization, for example dropout.

```
Message: look , next time get yourself some comfy shoes . you are gonna come back again with me , are not you ?
Reply: never thank you for inviting me .

Saving model as total best val loss decreased from inf to 249.583710
Epoch 1 --- 1141 sec: Loss 2.125520, val_loss: 2.062675
Saving model as total best val loss decreased from 249.583710 to 243.774765
Epoch 2 --- 1144 sec: Loss 1.741344, val_loss: 2.014668
Epoch 3 --- 1127 sec: Loss 1.544575, val_loss: 2.039159
Epoch 4 --- 1122 sec: Loss 1.381695, val_loss: 2.088749
Epoch 5 --- 1123 sec: Loss 1.258485, val_loss: 2.149965
Epoch 6 --- 1130 sec: Loss 1.161567, val_loss: 2.212848
Epoch 7 --- 1125 sec: Loss 1.085804, val_loss: 2.273186
Epoch 8 --- 1130 sec: Loss 1.022588, val_loss: 2.333526
Epoch 9 --- 1130 sec: Loss 0.971269, val_loss: 2.385126
Epoch 10 --- 1128 sec: Loss 0.929741, val_loss: 2.446960
Epoch 11 --- 1126 sec: Loss 0.894302, val_loss: 2.491678
Epoch 12 --- 1128 sec: Loss 0.864841, val_loss: 2.529509
Epoch 13 --- 1125 sec: Loss 0.839594, val_loss: 2.584446
Epoch 14 --- 1127 sec: Loss 0.817267, val_loss: 2.608792
Epoch 15 --- 1127 sec: Loss 0.800188, val_loss: 2.641543
Epoch 16 --- 1129 sec: Loss 0.783343, val_loss: 2.686161
Epoch 17 --- 1129 sec: Loss 0.768789, val_loss: 2.724887
Epoch 18 --- 1120 sec: Loss 0.756645, val_loss: 2.736274
Epoch 19 --- 1123 sec: Loss 0.745200, val_loss: 2.755772
Epoch 20 --- 1126 sec: Loss 0.735900, val_loss: 2.804235
Epoch 21 --- 1128 sec: Loss 0.726838, val_loss: 2.803058
Epoch 22 --- 1144 sec: Loss 0.718455, val_loss: 2.841672
Epoch 23 --- 1144 sec: Loss 0.711133, val_loss: 2.883457
Epoch 24 --- 1138 sec: Loss 0.705340, val_loss: 2.880698
Epoch 25 --- 1134 sec: Loss 0.700010, val_loss: 2.905796
Epoch 26 --- 1133 sec: Loss 0.693952, val_loss: 2.943427
Epoch 27 --- 1134 sec: Loss 0.690339, val_loss: 2.972523
Epoch 28 --- 1139 sec: Loss 0.687863, val_loss: 2.954038
Epoch 29 --- 1140 sec: Loss 0.682479, val_loss: 2.954602
Epoch 30 --- 1139 sec: Loss 0.677923, val_loss: 3.001948
Epoch 31 --- 1154 sec: Loss 0.675326, val_loss: 2.993776
Epoch 32 --- 1175 sec: Loss 0.671986, val_loss: 2.999511
Epoch 33 --- 1169 sec: Loss 0.669467, val_loss: 3.011023
Epoch 34 --- 1163 sec: Loss 0.666038, val_loss: 3.028480
Epoch 35 --- 1163 sec: Loss 0.665600, val_loss: 3.032686
```

Figure 19: Example of Sequence to Sequence model overfitting.

Reducing the Hidden State Dimensionality hyperparameter does reduce the complexity of the model, but it did not seem to have an impact on the overfitting problem. Adding varying levels of dropout did help slightly but not a great deal. Switching to a classic Bahdanau attention mechanism [3] did decrease the overfitting. Using this method the context vector is calculated using the previous decoder timestep instead of the current timestep and is concatenated with the input to the LSTM. It's not clear exactly why this decreased overfitting, but it's possible that concatenating the context vector to the decoder output caused the model to overfit. Even though this change allowed models to

reach a lower validation loss, all models eventually started to overfit after around the seventh epoch. Models stopped at this point would give very general and safe answers such as "I don't know" on almost every reply. For this reason, It seems that validation loss is not a good indicator for the quality of a chatbot. Other projects have confirmed this finding [15]. To combat this, each class of model was saved twice, once when it achieved the lowest validation loss and another after approximately two days of training or until the training loss stopped decreasing. The checkpoint which achieved the lowest validation loss is used for the automatic metrics whereas, the other checkpoint is used for the human metrics.

The Transformer hyperparameters specified in the design were out-performed on all metrics by a smaller sized Transformer. For this reason only the results for this tuned Transformer is used, which had the following differing hyperparameters:

- Hidden dimensionality d_{model}: 512
- Multi-Head Self Attention heads h: 8
- Query, Key Dimensionality d_k: 64
- Value Dimensionality d_v: 64
- Residual Dropout: 0.1

Hyperparameter tuning seems to be unavoidable for unique problems. The base parameters provided by studies provide a good starting point but they are too general to apply to every text generation problem. Unfortunately, for large scale problems like this, tuning can be very time consuming, especially if extensive computational power is not available.

To allow for reproducibility of the results found here, specific training details can be found below:

Model Name	**Batch Size**	**Epochs**	**Time Per Epoch (same hardware)**
Seq2Seq + Attention	64	40	2864s
Multiple Encoders	64	15	1070s
Transformer	256	80	315s

Table 2: Training Configurations

A link to download the trained models can be found on the project github page: https://github.com/psyfb2/Chatbot

5.3 Command Line Interface

The CLI is implemented in a file called main.py. Each model was implemented in a separate python file with functions for training, inference, saving and definitions of the model. Models are defined as a subclasses of the Tensorflow tf.keras.Model class. Custom layers which are subclasses of tf.keras.Layer were also often used in the model definitions. For instance, below is the constructor definition for the Transformer:

```
class Transformer(tf.keras.Model):
    ''' Q&A Transformer https://arxiv.org/pdf/1706.03762.pdf '''
    def __init__(self, d_model, num_layers, num_heads, d_ff,
                 input_max_pos, output_max_pos, vocab_size,
                 use_segment_embedding, dropout=0.1):
        super(Transformer, self).__init__()

        tied_embedding = Embedding(vocab_size, d_model)

        self.encoder = Encoder(num_layers, d_model, num_heads, d_ff,
                               tied_embedding, input_max_pos, use_segment_embedding,
                               dropout)
        self.decoder = Decoder(num_layers, d_model, num_heads, d_ff,
                               tied_embedding, output_max_pos, dropout)
```

```
self.output_layer = Dense(vocab_size)
```

Model files use a common text_preprocessing.py file, which implements the preprocessing procedures defined in the design. Functions in text_preprocessing are unit tested using the unittest framework. Depending on the train or talk arguments received, main.py will call the relevant functions from the model files. Another python file called eval.py is responsible for evaluating a trained model based on the automatic metrics and the functions in this file are called by main.py to evaluate a specified model. Unfortunately, a built-in beam search method is not available in Tensorflow, beamsearch.py implements this feature and can be used generally for any model.

5.4 Website

The website was implemented in accordance to the design, however Tensorflow Serving was not used because of a lack of funds. Maintaining a Tensorflow Serving docker is expensive unlike a single Flask server. This is because Tensorflow Serving was designed for large scale production systems. Under real world production settings, where funds are available, Tensorflow Serving would be used. Having said this, the models running on just a flask server perform well and still hit all the non-functional requirements.

Google Cloud App Engine was used to deploy the server. Below is a figure showing the front-end of the website:

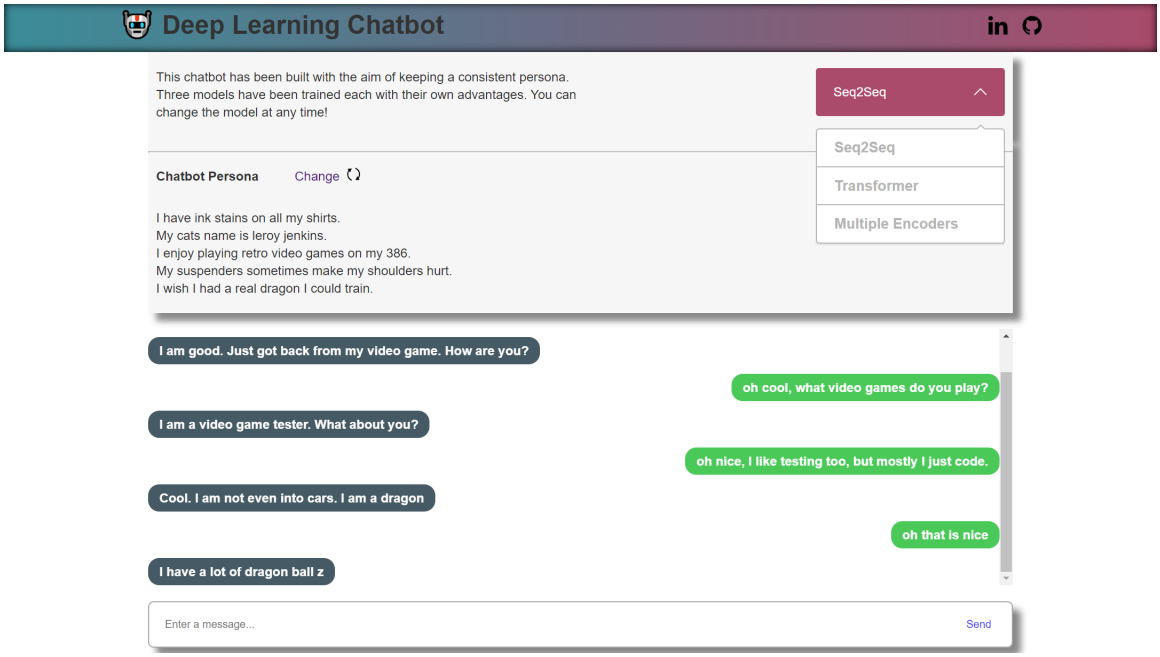

Figure 20: Website front-end.

The website provided some extra capabilities not planned for in the design. This includes choosing the model to interact with and allowing users to change the persona. The Flask backend provided an API endpoint which was interfaced with using Ajax on the front-end. This allowed replies to be loaded dynamically, without the need for any reloading. Changing the persona or model also does not require reloading.

The website did pass all of the non-functional requirements. (1) Reply length was kept under the target 5 seconds, (2) The website remained responsive when there was 5 or less concurrent users, (3) Google Cloud is a highly reliable platform so the up time should remain at 99% or higher, (4) The website has not been known to crash, all functionalities have been tested, (5) The website is mobile friendly and usable by devices with small screens.
A link to the live website can be found on the github page for this project.

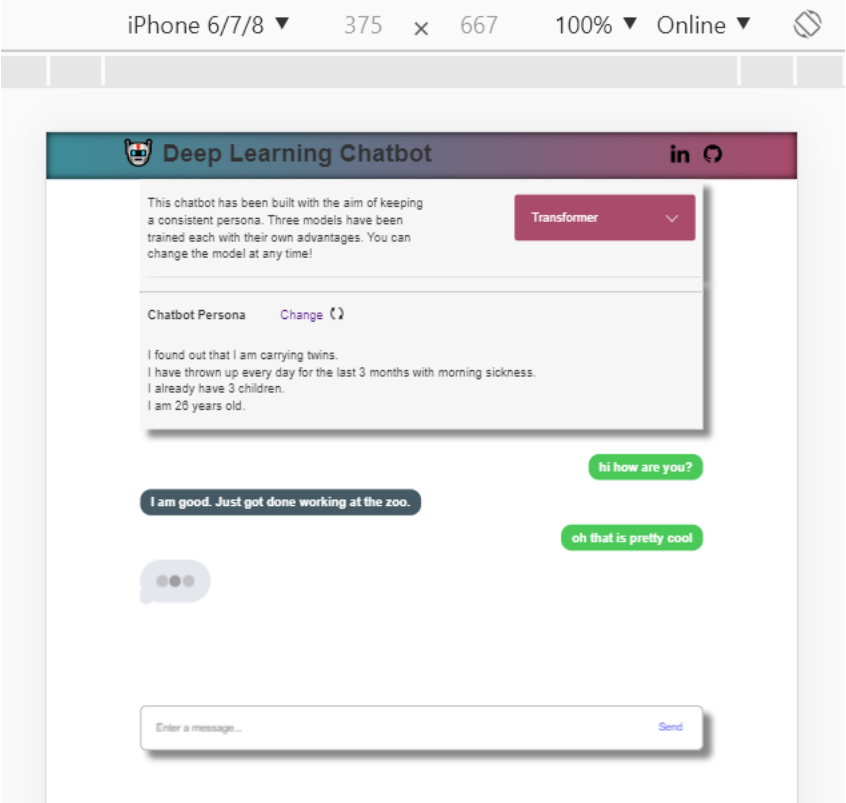

Figure 21: Example of how the website looks on an iphone. The page is responsive and resizes accordingly.

6 Results and Analysis

In this section we compare the performances of the trained models on various metrics. Ultimately, we seek to find a model which outperforms the other models. A model is ranked by taking it's minimum rank from Perplexity and F1. Results for the automatic evaluation metrics are shown in the table below:

Model Name	Perplexity	F1
Seq2Seq + Attention	**46.0**	**0.180**
Multiple Encoders	**44.3**	**0.171**
Transformer	46.3	0.140

Table 3: Perplexity: lower is better. F1: higher is better. Seq2Seq + Attention and Multiple Encoders come joint first.

It's hard to see a clear winner. On one hand, Perplexity is a measure of how unsure the model is of the correct response. Whereas, F1 measures word overlap between the predicted and correct response. All the models are quite tightly packed, so the human evaluation metrics will provide more clarity. Furthermore, Automatic metrics do not correlate well with human evaluation metrics [20]. Consequently, more weight will be placed on the human evaluation metrics for deciding the best model. Due to the COVID-19 outbreak, we were restricted in the number of participants who could take part. In total, eight people had a conversation with each model lasting ten utterances and were asked to score the models based on the evaluation metrics. The average score for each metric, for each model is shown in the table below. Models were ranked by performing a weighted sum $(2.5 * Fluency + 5 * Engagingness + 2.5 * Consistency) * Persona$. Generally, it is easy for a model to achieve high fluency and consistency, if the engagingness is low, as is the case for models stopped at the lowest validation loss. This is because the model can provide generic and boring answers which would result in high fluency and consistency such as "I'm not sure". However, this type of model can be viewed as cheating because it is not actually learning useful replies, only generic ones. The weighted sum above emphasizes the importance of interesting and diverse answers meanwhile, multiplying by the persona detection penalises models effectively for not maintaining the all important persona.

Model Name	Fluency	Engagingness	Consistency	Persona Detection
Seq2Seq + Attention	**2.16**	**3.28**	**2.59**	**1.00**
Multiple Encoders	2.27	2.50	3.75	1.00
Transformer	1.84	3.78	1.97	0.75
Human	4.64	4.07	4.60	1.00

Table 4: Seq2Seq came out on top in the human evaluation metrics.

This phenomenon was observed with the Multiple Encoders model. It can provide answers which strongly emphasize it's persona, but it is very predictable, usually replying with "I like to do x" where "x" is included in its persona and when it makes sense to reply with this. For this reason it suffered the lowest engagingness. This is likely because of the more explicit inclusion of the persona into the model, within the attention mechanism and the dedicated persona encoder. This allowed it to focus more on the persona, but in turn it became predictable in it's answers. Conversely, the Transformer produces some entertaining replies and seems to be quite spontaneous. However, it struggles to link its replies with its persona which can be seen by its lowest consistency and persona detection score. The most likely explanation for this, is that the attention heads have learnt to pay more attention to the message segment when compared to the persona segment. It's not clear why this has happened, perhaps this is a more efficient method of optimising the loss function when solely using Dot-Product attention. Sequence to Sequence with Attention appears to bring the best of both worlds. It produces fairly natural responses, meanwhile maintaining a consistent persona. Anecdotally, the majority of participants felt that Sequence to Sequence was overall the more robust model out of the three. Below are some example conversations from the human evaluation stage:

Persona
I like reading mystery novels.
Going to the beach is my favorite past time.
I love pizza.
I am an american citizen.
my son is a football player.
[HUMAN] do you like pizza ?
[BOT] I do love pizza what is your favorite ?
[HUMAN] oh cool , I like vegitarian pizza mostly , but fast food is not good for you .
[BOT] I am a big fan of chili peppers
[HUMAN] what other foods do you like ?
[BOT] I like all kinds of food
[HUMAN] ah ok that is cool , do you have any hobbies ?
[BOT] yes , I love going to the beach . you ?

Table 5: Example conversation with Sequence to Sequence + Attention.

Persona
i am married to my wonderful husband .
i am a woman .
i own my own small marketing consulting agency .
my favorite band is radiohead .
we have two dogs small dogs .
[HUMAN] hi , how are you day ?
[BOT] i am good . just got done with my wonderful dogs
[HUMAN] oh nice , how many dogs do you have ?
[BOT] i have 2 dogs . do you have a favorite band ?
[HUMAN] cool , not really I like listening to rap music more .
[BOT] what do you do for a living ?
[HUMAN] well at the moment i'm a student at the university of nottingham studying computer science , what about you ?
[BOT] i am a business owner of a business consulting agency .
[HUMAN] well that is amazing , are you rich ?
[BOT] i am . are you married ?

Table 6: Example conversation with the Multiple Encoders model.

Persona
i work at home , editing .
i have a cat .
i have depression and anxiety so i do not really go out a lot .
i hope to move out soon .
i live at home with my dad and brother .
[HUMAN] hi , how are you day ?
[BOT] i am good . just got done with my shift at aldi and getting ready for work .
[HUMAN] oh that is cool , so you have two jobs then ?
[BOT] i have a job in nyc saw some people doing crack on the way there
[HUMAN] lol that is really crazy , is there a lot of drugs in new york ?
[BOT] i do not think so what do you do for a living ?
[HUMAN] I study computer science at the university of nottingham , how about you ?
[BOT] i do not have time for that , i study a lot . what are your hobbies ?
[HUMAN] mostly just football and climbing , what about you ?
[BOT] i am in school studying to be a lawyer but i bartend on weekends

Table 7: Example conversation with the Transformer model.

Below, we will specifically test the models ability to keep a consistent persona. Clearly, the Seq2Seq + Attention and Multiple Encoders models are good at keeping a consistent set of background facts. The Transformer however, cannot seem to do this. Having said this, the Transformer has great potential. It is an entertaining model with high engagingess, if a stronger sense of persona could be incooperated into the model, then it is likely to outperform the other models.

Persona sentence to test
my dogs name is toto.
[HUMAN] do you have a dog?
[SEQ2SEQ] i do, i have a dog named toto. what about you?
[MULTIPLE ENCODERS] i do, i have a dog named toto. what about you?
[TRANSFORMER] i have a cat named radar
[HUMAN] what is your dogs name?
[SEQ2SEQ] his name is toto whats yours?
[MULTIPLE ENCODERS] his name is toto whats yours?
[TRANSFORMER] my favorite is coffee with cheese
[HUMAN] what pets do you have ?
[SEQ2SEQ] A golden retriever. He is a great dane
[MULTIPLE ENCODERS] yes i have a dog named toto
[TRANSFORMER] i have a lab named sirius

Table 8: Testing to see whether models can stay consistent with their persona. Multiple Encoders and Seq2Seq do well, the Transformer does not produce a consistent set of facts.

Persona sentence to test
i am a school psychologist.
[HUMAN] what is your job?
[SEQ2SEQ] i am a school psychologist
[MULTIPLE ENCODERS] i am a school psychologist
[TRANSFORMER] i am a student. i like to be in my own company though.
[HUMAN] what do you do for a living my friend?
[SEQ2SEQ] i am a psychologist, i am learning to play piano.
[MULTIPLE ENCODERS] i am a psychologist, i am learning to play piano.
[TRANSFORMER] i am a stay at home mom. taking care of the kids is full time job. you?
[HUMAN] are you a butler?
[SEQ2SEQ] no i am not. do you play any instruments ?
[MULTIPLE ENCODERS] no i am not. do you play any instruments ?
[TRANSFORMER] i have a ton of pet pugs

Table 9: Again Seq2Seq and Multiple Encoders keep a consistent persona. However, they do not always produce the same answers. The Transformer model fails to keep a consistent persona.

This further confirms the fact that Sequence to Sequence and Multiple Encoders models can keep a consistent persona. Whereas, the Transformer rarely does this. Attention plots for the models used here are not as intuitive as those found in Neural Machine Translation. It's not obvious exactly what the models are paying attention too. Below are some attention plots for the three models:

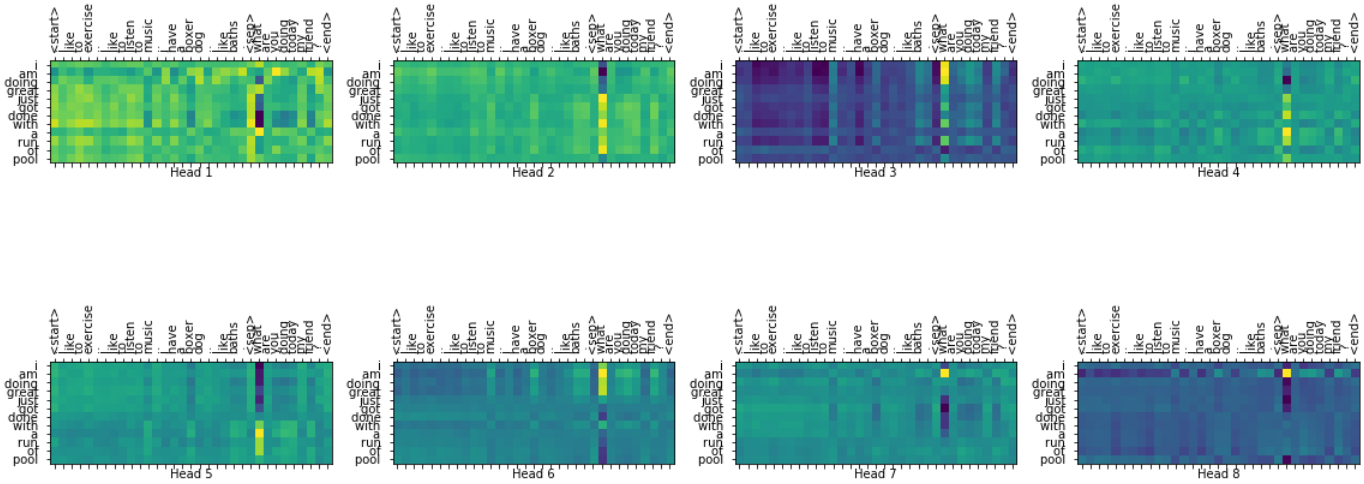

Figure 22: Attention Plot over 8 heads of the last decoder layer of Multi-Head Self Attention. Most of the heads seem to be paying attention to the input word 'what'.

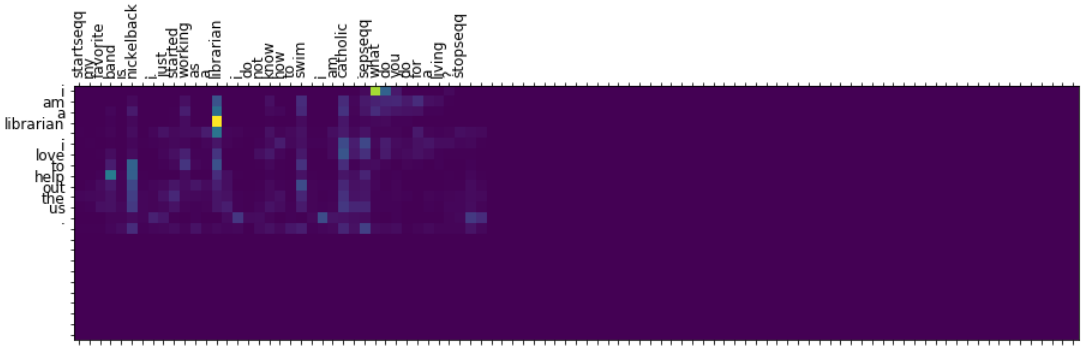

Figure 23: Attention Plot of Seq2Seq model. The output word 'I' is paying attention to the input word 'what' and the output word 'librarian' is paying attention to the input word 'librarian' which is part of the persona. This suggests that attention is attending to the relevant parts of the persona when it is needed.

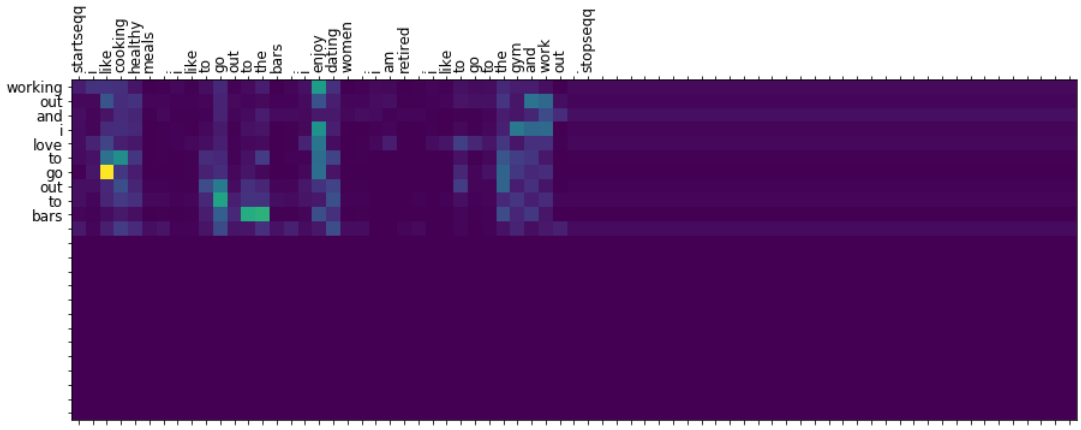

Figure 24: Attention Plot of Persona encoder for the Multiple Encoders model. The output word 'bars' is attending to the words 'to the bars' in the input persona.

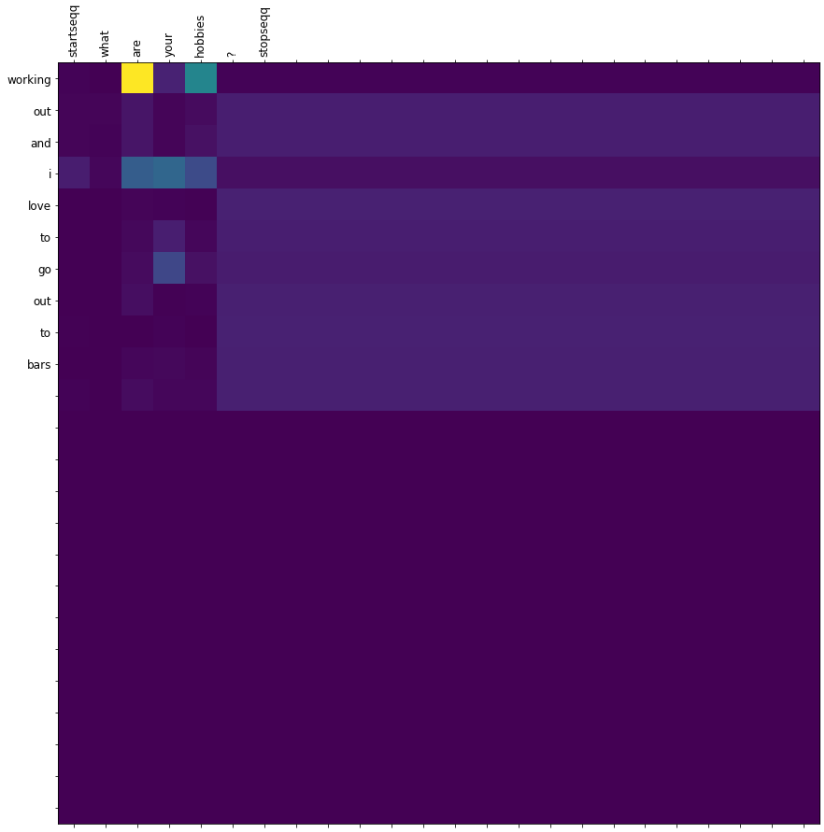

Figure 25: Attention Plot of Message encoder for the Multiple Encoders model. The word 'working' is attending to the words 'are' and 'hobbies'. This suggests the model has identified 'working out' as an answer to a question about 'hobbies'.

To conclude, the Sequence to Sequence + Attention outperforms the other models on both automatic and human evaluation metrics. Multiple Encoders is a good model but it overly emphasizes the persona causing its responses to become predictable. The Transformer seems to be a good chatbot in general, but cannot identify with the main goal of this project. Overall, the project has been successful, all requirements have been achieved including the end goal of solving the problem of keeping a consistent persona. Future improvements which could be made are outlined in the next section.

7 Reflection

This section provides an overview of the project as a whole from the authors perspective.

7.1 Project Management

Originally, the plan was to train a different set of models. Due to the fact that more research could be conducted, it was found later in the project that other models are more suited for the problem at hand. This was dealt with by working using an agile approach. Every set of two weeks, a sprint was planned in order to achieve a milestone, for example training a model. Below is the Gaant chart for the original plan:

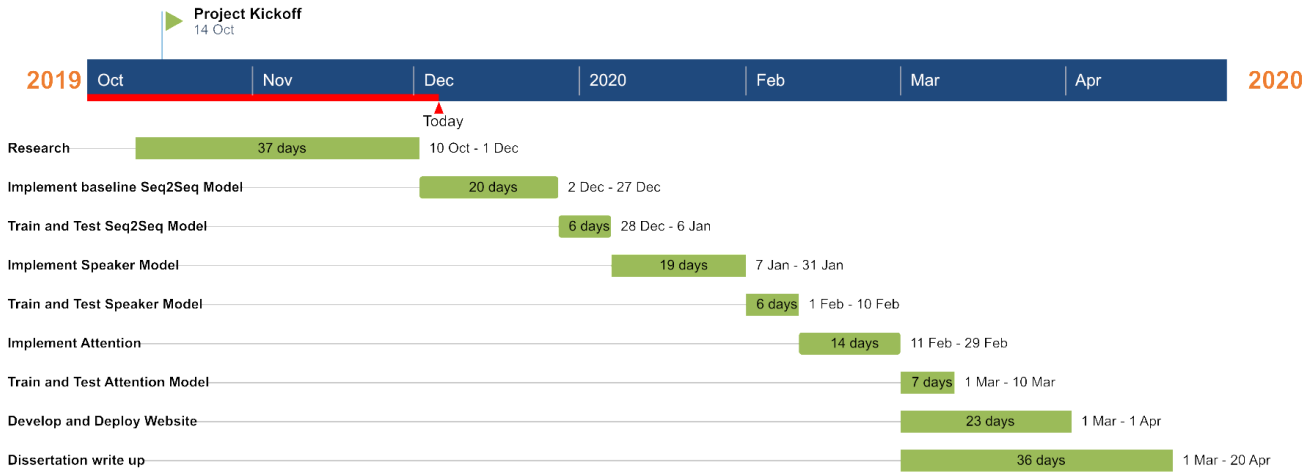

Figure 26: Original plan as a Gaant Chart.

The plan was not followed exactly. Because of the limited time at hand it was decided state-of-the-art models such as Transformers should be implemented instead of the Speaker Model. On top of this, the time taken to implement the models was severely underestimated. Before the start of this project I had little experience with python and machine learning, it was a huge learning curve. Furthermore, machine learning models are extremely hard to debug, a lot of the time they won't work as expected and the reason why is unclear. An example of this occurred when implementing the Transformer. It would always give the same reply no matter the input message. In the end the bug was using the wrong activation function in the fully connected layer, but this required cross-checking papers against my own implementation to ensure everything is correct, which can be very time consuming. Utilising an agile methodology helped greatly with dealing with setbacks and changing plans.

The project planning also could have been better. Implementation started in late December, which was not enough time in hindsight to comfortably implement all the models. Overall, this project has been a massive learning experience. I am now comfortable with a lot of the maths and implementation details behind machine learning algorithms. On top of this, the ability to read and understand scientific papers has increased and web development skills were refreshed.

7.2 Contributions

The project was successful, it achieved it's main goal of creating a consistent persona and hit the other requirements along the way. However, the models still have a lot of room for improvement. This work maybe useful to researchers

because they can see exactly which models outperform which, for this task and why. They could build on this work to produce more sophisticated models. Companies and researchers can also use the models trained here for their own experiments as the code has been made open source. We have also discovered that validation loss is not a good metric for early stopping. It is a very one dimensional measure, it does not take into account the diversity of responses which a model is able to produce, which is an extremely important attribute for good engagingness.

The project also provided novelty, with an unseen model tried for the task at hand. Many possible improvements can be made in order to advance the overall quality of chatbots while keeping a consistent persona.

- **Better Loss Measure:** Validation Loss is not a good indicator of chatbot quality. New measures need to be developed. this new measure should take into account how close the predictions are to the labels like before, but also the diversity of responses generated by the model.
- **Injecting Persona Vector:** All the models used in this project do not explicitly create a persona vector, which could be inserted into the model directly with the hidden state for example. This will likely further increase consistency and persona detection, while maintaining enganginess and fluency.
- **Improve Transformer:** The Transformer is an extremely promising model. It can produce very interesting responses, but it crucially lacks consistency. New methods could be developed in order to make the Transformer stronger in this area. An explicit, learnt persona vector could be injected, or maybe a separate encoder for the persona.
- **Recent Dialog History:** The models trained here only received the persona and current message as input. They had no knowledge of previous conversation history. This prevented the fluency of all models from being very high. The models can be improved by giving them a sense of the recent conversation history. The simplest way of doing this is to provide the persona, followed by the previous human message, followed by the previous bot response, followed by the current human message.

8 Bibliography

References

[1] Amazon. *Alexa Prize*. 2018. URL: https://developer.amazon.com/alexaprize.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *CoRR* abs/1409.0473 (2014).

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. *Learning Long-Term Dependencies with Gradient Descent is Difficult*. 1994.

[5] Denny Britz et al. "Massive Exploration of Neural Machine Translation Architectures". In: *CoRR* abs/1703.03906 (2017). arXiv: 1703.03906. URL: http://arxiv.org/abs/1703.03906.

[6] Rich Caruana, Steve Lawrence, and C. Lee Giles. "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping". In: *Advances in Neural Information Processing Systems 13*. Ed. by T. K. Leen, T. G. Dietterich, and V. Tresp. MIT Press, 2001, pp. 402–408. URL: http://papers.nips.cc/paper/1895-overfitting-in-neural-nets-backpropagation-conjugate-gradient-and-early-stopping.pdf.

[7] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[8] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[9] Cristian Danescu-Niculescu-Mizil and Lillian Lee. "Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs." In: *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics, ACL 2011*. 2011.

[10] John L. Gustafson. "Moore's Law". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1177–1184. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_81. URL: https://doi.org/10.1007/978-0-387-09766-4_81.

[11] Michiel Hermans and Benjamin Schrauwen. "Training and Analysing Deep Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., 2013, pp. 190–198. URL: http://papers.nips.cc/paper/5166-training-and-analysing-deep-recurrent-neural-networks.pdf.

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[13] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836. URL: http://arxiv.org/abs/1609.04836.

[14] Vid Kocijan et al. "A Surprisingly Robust Trick for the Winograd Schema Challenge". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 4837–4842. DOI: 10.18653/v1/P19-1478. URL: https://www.aclweb.org/anthology/P19-1478.

[15] Richard Kriszti. *Deep Learning Based Chatbot Models*. 2019. URL: https://arxiv.org/pdf/1908.08835.pdf.

[16] Jiwei Li et al. "A Diversity-Promoting Objective Function for Neural Conversation Models". In: *CoRR* abs/1510.03055 (2015). arXiv: 1510.03055. URL: http://arxiv.org/abs/1510.03055.

[17] Jiwei Li et al. "A Persona-Based Neural Conversation Model". In: *CoRR* abs/1603.06155 (2016). arXiv: 1603.06155. URL: http://arxiv.org/abs/1603.06155.

[18] Yanran Li et al. "DailyDialog: A Manually Labelled Multi-turn Dialogue Dataset". In: *CoRR* abs/1710.03957 (2017). arXiv: 1710.03957. URL: http://arxiv.org/abs/1710.03957.

[19] Zachary Chase Lipton. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: *CoRR* abs/1506.00019 (2015). arXiv: 1506.00019. URL: http://arxiv.org/abs/1506.00019.

[20] Chia-Wei Liu et al. "How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation". In: *CoRR* abs/1603.08023 (2016). arXiv: 1603.08023. URL: http://arxiv.org/abs/1603.08023.

[21] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *CoRR* abs/1508.04025 (2015). arXiv: 1508.04025. URL: http://arxiv.org/abs/1508.04025.

[22] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[23] MarketsAndMarkets. "Natural Language Processing Market by Component, Deployment Mode, Organization Size, Type, Application (Sentiment Analysis and Text Classification), Vertical (Healthcare and Life Sciences, and BFSI), and Region - Global Forecast to 2024". In: Curran Associates, Inc., 2019. URL: https://www.marketsandmarkets.com/Market-Reports/natural-language-processing-nlp-825.html.

[24] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[25] Kishore Papineni et al. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://www.aclweb.org/anthology/P02-1040.

[26] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation". In: *In EMNLP*. 2014.

[27] Martin Popel and Ondrej Bojar. "Training Tips for the Transformer Model". In: *CoRR* abs/1804.00247 (2018). arXiv: 1804.00247. URL: http://arxiv.org/abs/1804.00247.

[28] Ofir Press and Lior Wolf. "Using the Output Embedding to Improve Language Models". In: *CoRR* abs/1608.05859 (2016). arXiv: 1608.05859. URL: http://arxiv.org/abs/1608.05859.

[29] Prajit Ramachandran, Peter Liu, and Quoc Le. "Unsupervised Pretraining for Sequence to Sequence Learning". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 383–391. DOI: 10.18653/v1/D17-1039. URL: https://www.aclweb.org/anthology/D17-1039.

[30] Alan Ritter, Colin Cherry, and William B. Dolan. "Data-Driven Response Generation in Social Media". In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*. Edinburgh, Scotland, UK.: Association for Computational Linguistics, July 2011, pp. 583–593. URL: https://www.aclweb.org/anthology/D11-1054.

[31] Herbert E. Robbins. "A Stochastic Approximation Method". In: 2007.

[32] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pp. 65–386.

[33] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[34] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0893608014002135.

[35] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *CoRR* abs/1508.07909 (2015). arXiv: 1508.07909. URL: http://arxiv.org/abs/1508.07909.

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *CoRR* abs/1409.3215 (2014). arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.

[37] Joseph Turian, Luke Shen, and I. Melamed. "Evaluation of Machine Translation and its Evaluation". In: (Sept. 2003).

[38] A. M. Turing. "Computing Machinery and Intelligence". English. In: *Mind*. New Series 59.236 (1950), pp. 433–460. ISSN: 00264423. URL: http://www.jstor.org/stable/2251299.

[39] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[40] Oriol Vinyals and Quoc V. Le. "A Neural Conversational Model". In: *CoRR* abs/1506.05869 (2015). arXiv: 1506.05869. URL: http://arxiv.org/abs/1506.05869.

[41] Joseph Weizenbaum. "ELIZA - a computer program for the study of natural language communication between man and machine." In: *Commun. ACM* 9.1 (1966), pp. 36–45. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm9.html#Weizenbaum66.

[42] P. J. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 1558-2256. DOI: 10.1109/5.58337.

[43] Saizheng Zhang et al. "Personalizing Dialogue Agents: I have a dog, do you have pets too?" In: *CoRR* abs/1801.07243 (2018). arXiv: 1801.07243. URL: http://arxiv.org/abs/1801.07243.