A Generative Chatbot with Natural Language Processing

David Liebman david.c.liebman@gmail.com

February 22, 2020

Coordinator 1 Coordinator 2

Department of Computer Science

SUNY New Paltz

Abstract

Our goal in this thesis is to create a chat-bot, a computer program that can respond verbally to a human in the course of simple day-to-day conversations.

We use a deep learning neural network model called the Transformer to develop the chat-bot. A full description of a Transformer is provided. We chronicle the use of a few different Transformer-based Natural Language Processing models to develop the chat-bot, including the Generative Pre-training Transformer 2 (GPT-2). For comparison we include a Gated Recurrent Unit (GRU) based model. Each of these are explained below.

We are also interested in installing the chatbot code on a small device such as the Raspberry Pi with speech recognition and speech-to-text software. In this way we might be able to create a device that can carry out a verbal conversation with a human. For the GRU-based model we can use a Raspberry Pi 3B with 1GB RAM. A Raspberry Pi 4B with 4GB of RAM is needed to run a chatbot with the GPT-2.

Contents

List of Figures

List of Tables

Chapter 1

Background/History of the Study

1.1 Background

In their paper Vinyals et al (2015)[1] discuss making a chatbot using a neural network configured for sequence to sequence neural machine translation. We code our own sequence to sequence chatbot, though our results are less than spectacular. As our hand coded model does not run sufficiently well, we use code authored by Inkawhich et al (2018)[2].

In their paper Vaswani et al (2017)[3] discuss using the Transformer architecture for solving machine learning tasks. We train a transformer model as a chatbot.

Also Radford et al (2019)[4] discuss the 'Generative Pre-training Transformer 2' neural network for NLP tasks. The Generative Pre-training Transformer 2 model is based largely on the Transformer architecture.

We implement a chatbot with a Generative Pre-training Transformer 2 model. We use a program library from Wolf et al (2019)[5] to run our model.

It is worth noting that with the appearance of the Transformer architecture and WordPiece vocabulary scheme, some technologies have become redundant or obsolete. This may be true of any model that uses Recurrent Neural Network components and also the traditional word vector embeddings.

1.2 Recurrent Neural Network Components

Overview

The goal behind Recurrent Neural Network components is to detect patterns. Here we explore a simple RNN.

The simplest Recurrent Neural Network has two inputs and two outputs. They can be arranged in chains. In our example the input will be a sequence of data and the Recurrent Neural Network chain will be a line of components of the same length as the data.

One input from each component is the hidden state output from the Recurrent Neural Network to the left. Another input is the current input from the sequence that the chain is monitoring. One output is the generated hidden state, meant for the component to the right. The last output is the value that the Recurrent Neural Network outputs or surmises.

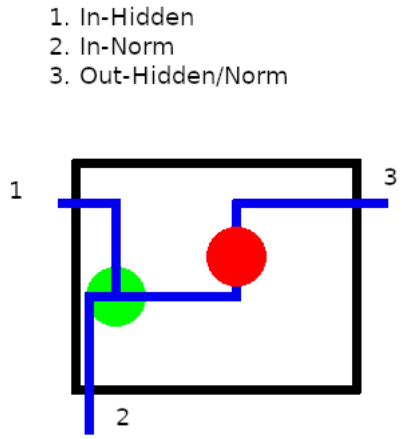

Figure 1.1: RNN - 1 and 2 are inputs, 3 is the output.

In our diagram the two inputs are labeled on the left side, and the single output does double duty as both the hidden state output and the value that the Recurrent Neural Network outputs or surmises.

There are several designs for a Recurrent Neural Network component. The inner workings of these components are what makes them different. In the example in the diagram the inner workings are very simple. Two paths, labeled as inputs, take data into the Recurrent Neural Network. Their data is combined in the green circle. This combination is done with concatination and simple feed forward neural network components. The output from the concatination is passed through the red circle. This is a 'tanh' activation operation that limits the output to values from -1 through 1. This 'tanh' activation keeps the output within reasonable values. Finally the output is delivered outside the component to the program employing the Recurrent Neural Network. In this diagram there is only one output. The single output would serve as both the hidden state output for that position in the chain, and also the data output component for that position in the chain.

Gated Recurrent Unit

An implementation of a Recurrent Neural Network is the 'Gated Recurrent Unit'. A GRU has two inputs and two outputs. The formulas for a Gated Recurrent Unit, as outlined by Denny Britz in the website WILDML (Britz et al 2015)[6], are as follows.

$$z = \sigma(x_t U^z + s_{t-1} W^z)$$

$$r = \sigma(x_t U^r + s_{t-1} W^r)$$

$$h = tanh(x_t U^h + (s_{t-1} \circ r) W^h)$$

$$s_t = (1 - z) \circ h + z \circ s_{t-1}$$

The Gated Recurrent Unit has two inputs and two outputs. It also has two internal gates. One internal gate is the 'reset' gate. This one determines how much of the previous input is combined with the new value calculated by the mechanism of the Gated Recurrent Unit. It is denoted as 'r' above. Another internal gate is the 'update' gate. The update gate decides how much new information is to be included in gate computation. It is denoted as 'z'.

Here 's_t' is the symbol for the output. The two inputs are 'x_t' and 's_{t-1}' . 'x_t' is the hidden state input. 's_{t-1}' is the regular input for the Recurrent Neural Network or Gated Recurrent Unit. Sigmoid activation is used on the two gates, while tanh activation is used to compute the hidden output. The hidden output is found in the third line, denoted as 'h'.

In the last line, the regular output is determined using the 'dot' product which is denoted with a circle, along with an addition operation. In the two gate formulas (the first and second) the output is determined as the sum of two matrix multiplication operations passed through sigmoid activation. This produces values in the range of 0 to 1.

Under most programming circumstances the Gated Recurrent Unit is not implemented by the average programmer. The programmer employs a language like python and a programming library like Pytorch or Tensorflow. The library then implements the Gated Recurrent Unit and makes it easy for the programmer to use that implementation.

1.3 Sequence to Sequence

Translating text from one language to another has become a common task for computers. The Sequence to Sequence architecture is often used today for this purpose.

Here, with a few diversions, we explain how this works.

A naive approach to translation involves using a dictionary. You would encode each key as a word from one language and the value for that key would be the translated word in the target language.

Of course this doesn't work, because different languages not only have different words for the same thing, but they also have different sentence structures for what might be similar concepts.

A better approach is sequence to sequence translation. A description follows with a section at the end for how training works.

In this approach we use recurrent neural networks to obtain our translation. Two recurrent neural network components are employed. One is responsible for the input language and the other for the output.

Formula

Below is a product formula. It says that the y_n term relies on the sequence of terms that starts with y_1 and continues to y_{n-1}. It also has v that represents the 'thought vector' which is present along with $y_1, ..., y_{t-1}$ as an input to the decoder Gated Recurrent Unit and helps formulate y_t.

On the left side of the formula we have $x_1, ...x_T$ which represents the input sequence for the encoder. Also we have $y_1, ..., y_{T'}$ which represents the entire output. T is the length of the input while T' is the length of the output. Those two T terms do not need to be the same, and often are not. Again, v is the hidden state passed from the encoder to the decoder as a 'thought vector'.

$$p(y_1, ..., y_{T'}|x_1, ..., x_T) = \prod_{t=1}^{T'} p(y_t|v, y_1, ..., y_{t-1})$$

Every prediction relies on all the previous predictions up until that point. This is what happens in sequence to sequence and transformer architectures when the model is producing output.

The probability for the entire output is the same as the product of the probability of each word as it is generated.

This function can be found in Sutskever et al (2014)[7].

Word Embeddings

Also employed are two vocabulary sets. One vocabulary is for the source language and another is for the target language. A table of word vectors the size of the input vocabulary is created and a maximum sentence length is picked. There is also a 'hidden size', which is an important dimension in the model. In practice the hidden size could be 300 units and more for this kind of application.

Words are translated from strings to individual numbers from a vocabulary dictionary. The dictionary only contains a single unique number for every word. Then the number is passed through an embedding structure. This turns the single number into a vector of numbers that is the same size as the RNN hidden dimension. Then, from that time on the model uses the vector instead of words.

The contents of the embedding unit is a table of numbers, all are of the size of the RNN hidden dimension. The vectors are usually, but don't have to be, unique values. There is one complete hidden dimension sized vector for each word in the vocabulary.

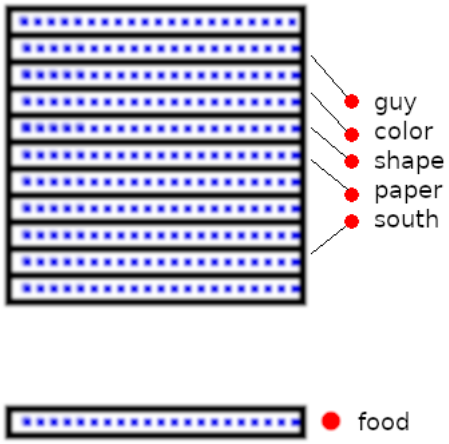

Figure 1.2: Embeddings - Each word from a dictionary is converted to a vector of numbers.

The vectors can be initialized randomly or they can be filled with predetermined values. As the network trains the embedding values can either be modified or frozen in place. Typically if the contents were initialized randomly the values would be trained. If the contents were filled with predetermined values you don't want to train them or change them in any way.

There are at this writing two main types of pretrained word embeddings. One is called 'Word2Vec' and one is called 'GloVe'.

Word2Vec is short for 'Word to Vector.' (Mikolov et al, 2013)[8] GloVe is short for 'Global Vector.' (Pennington et al, 2014)[9] .

Word vectors are created composed of floating point numbers. Each word in the vocabulary is translated to an integer which remains the same throughout the use of the translator. The word vectors are arranged in a table of floating point numbers with one dimension being the size of the

input vocabulary and one dimension being the hidden size for the vector.

Corpus

A text must be prepared for training. A text corpus with source and target pairs is chosen. Sentences in the source corpus are paired with sentences with the same meaning in the target corpus. Sentence length is observed and for all sentences shorter than that length a special 'end-of-sequence' token is appended to all sequences in both languages.

Recurrent Elements

The recurrent unit in this case is a Gated Recurrent Unit. The Gated Recurrent Unit takes as input a single vector. It processes the vector and returns another vector. This could be exactly the same as the input but is usually somehow changed. The input vector and the output vector have the dimension of the 'hidden size' mentioned above. Throughout the discussion of this model the hidden size will remain the same. The Gated Recurrent Unit also operates on two hidden states. One hidden state, a vector, is taken in and another hidden state, also a vector, is produced for output.

So far the model takes a word, translates it to an integer, and finds the vector in the word embedding table that represents that word. It does this for the first word and all subsequent words one by one. Then it gives the entire vector for a word to the GRU. The GRU takes the word and passes it to some inner components. It decides weather to return as output just the input or the input modified. This is what the Gated Recurrent Unit does internally.

We will use the metaphor of a chain when we describe input and output segments that are composed of Gated Recurrent Units. The input chain is sometimes referred to as the Encoder, and the output chain is sometimes referred to as the Decoder.

Encoder

The input segments, composed of Gated Recurrent Units, take two input vectors and return two output vectors. One input is the vector from the embedding table. Another input vector is the hidden state. The hidden state is the same size as the input from the embedding table, but typically it comes from the previous input Gated Recurrent Unit. The output vector is a hidden value.

The very first Gated Recurrent Unit in the input chain ignores the fact that the first word has

no hidden value. It consumes the first word vector. Then it passes it's output to the next Gated Recurrent Unit in the chain. This Gated Recurrent Unit uses the output of the previous Gated Recurrent Unit as the hidden value. It also uses the vector for the second word. It passes it's important information to the Gated Recurrent Unit to it's right. Then the last Gated Recurrent Unit in the chain passes it's hidden state to the output chain, the decoder.

A complicating detail is that although many Gated Recurrent Units are called for in the input chain they all use the same set of weights and biases. For this reason only a single input Gated Recurrent Unit is used for all of the words in practice. Outputs are calculated and then cycled around and fed with the next word from the sentence in vector form to the input of the Gated Recurrent Unit.

Decoder

The output chain is in charge of generating tokens that represent, in this case, the translation of the input in the output language. The output uses Gated Recurrent Unit segments also. The first hidden input for the first output cell is taken from the last hidden output of the last recursive unit of the input chain. It is important because it is the spot where a great amount of data is passed from the input chain to the output chain. The connection at this point is said to carry the 'thought vector'. Most of the information responsible for translating one language to another is passed at this point.

The hidden values from the input section are passed to the first output Gated Recurrent Unit. It outputs the values that are later converted to the first word of the output. The first output Gated Recurrent Unit also has a hidden state. It passes the first word and the hidden state on to the second Gated Recurrent Unit.

The second Gated Recurrent Unit generates the second word and also its own hidden state. The second word is recorded and the word and the hidden state are passed on to the next Gated Recurrent Unit. This is repeated until a special 'end-of-sequence' token is found or until the number of tokens equals the maximum number allowed.

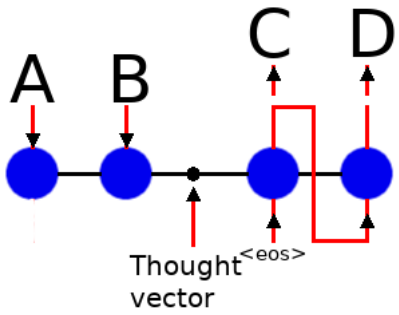

Figure 1.3: Seq2seq: A and B represent an input sequence and C and D represent the corresponding output.

In this figure we generalize a sequence to sequence model. The idea is that A and B, on the left side of the diagram, deal with the encoding of sentences. A and B would be consecutive words in a sentence, and the round blue nodes below A and B are Recurrent Neural Network units. C and D are outputs and in the right side of the diagram the blue nodes represent the output Recurrent Neural Network units. Between the input and the output there is a corridor of information exactly the size of the Recurrent Neural Network hidden vector.

All of the information that the decoder uses for it's output is present in this corridor and is passed along the corridor from the encoder. For this reason we refer to it as the thought vector.

Making this vector larger by increasing the size of the hidden unit size allows for more information in the thought vector. Size also increases the time to train the network. The size must also match the dimension of the vectors in the GloVe or Word2Vec download if one of those is used.

Ultimately exceedingly large hidden dimension does not improve the sequence to sequence model.

Output Tokens

Each output we have is currently in the form of a vector. These vectors are a long string of floating point numbers, each one the dimensions of the 'hidden size' mentioned above. What's done with them is they are converted to the dimensions of the output vocabulary. Then they are processed in what is called an 'arg-max' function. This processing determines the index of the maximum value in the new vocabulary sized vector. This index allows the program to look up the corresponding word in the output vocabulary. This word is then used as the model output at that point in the output chain.

How does the model know what target vocabulary indexes go at what point in the output? For this we refer back to the corpus that we already have. One part of the corpus is the source language. The other part is the target language. A sentence is applied to the input and for that sentence an output is generated. The output is called a prediction. If the target and the prediction are not the same, the model must modify itself so that in the future they are.

There are some inherent problems. Because the output of a Gated Recurrent Unit is constantly being reused as the input, lots of data that might be useful to have is lost when the Gated Recurrent Unit churns through internal operations where several matrix multiplication operations are performed together on input. With every iteration more data is lost and so, for example, the effective length of the input and output sentences must be short.

Another problem is that all input to be translated to target output has to be boiled down and passed to the output section through a small corridor the size of a typical word vector. This channel is sometimes referred to as the 'thought vector.' Somehow all necessary information must be there. This also limits the length of the input and output vectors. It does help, when setting up a model for later training, to make the hidden size larger, but it only helps so much. There is a point at which the benefit of increasing the hidden size is lost.

This is how some computer models do language translation. Using 'arg-max' is an example of a greedy approach. Another approach might use something like 'Beam Selection' but we're not going to get into that here.

1.4 Loss and Accuracy

At first the prediction from a model is not very close to the desired output. The output is compared to the prediction and a 'loss' is generated. 'Loss' measures the difference between the predicted output and the target. A larger loss represents two values, prediction and target, that are further apart.

Another metric is Accuracy. 'Accuracy' is a numerical representation of the difference between the desired output and the generated prediction. It is a percentage of the time that the output is exactly correct.

Getting a prediction, running input data through a neural network, is forward propagation. Training, then, is a mathematical process involving backpropagagtion. Backpropagation identifies

areas of the model weights that need to be modified in order to get the proper prediction in the future.

We take the derivative of the loss function in order to backpropagate. The derivative is manipulated with the learning rate. The original weight value is changed minutely. The amount changed with every backward propagation is dependent on the learning rate. The result is a set of adjusted weight matrices and a new loss. When these matrices are used later they allow for better predictions.

This is done over and over with every source/target sentence pair. Slowly the model is changed and predictions start to match the target. That's training. The loss should decrease over time and the accuracy should increase.

There are several numerical metrics that we can record during training that tell us how our model is training. The loss, a mathematical calculation of the difference between the model's output and the predicted value, is mentioned above. Loss is an important number. Also accuracy is important. Accuracy is a mathematical calculation of the difference between the model's output and the value that output should be, but it focuses on the number of times the model comes out with a correct prediction verses how many output values there are in total.

1.5 Attention Mechanism

Here we consider a simple attention mechanism that is used in the Sequence to Sequence model by Inkawhich et al (2018)[2]. The concept for this attention comes from Luong et al (2015)[11].

Luong et al (2015)[11] are interested in three kinds of calculation for their attention mechanism. The three methods use slowly increasing levels of complication. First they propose a method that just uses the dot product. Then they propose a method that just uses a field of weights. Finally they use a method that uses concatination, along with a field of weights and a pass through a 'tanh' activation layer.

$$\boldsymbol{score}(\boldsymbol{h_t}, \bar{\boldsymbol{h}}_{\boldsymbol{s}}) = \begin{cases} \boldsymbol{h_t}^{\intercal} \bar{\boldsymbol{h}}_{\boldsymbol{s}} & \text{dot} \\ \boldsymbol{h_t}^{\intercal} \boldsymbol{W_a} \bar{\boldsymbol{h}}_{\boldsymbol{s}} & \text{general} \\ \boldsymbol{v_a}^{\intercal} \tanh(\boldsymbol{W_a}[\boldsymbol{h_t}; \bar{\boldsymbol{h}}_{\boldsymbol{s}}]) & \text{concat} \end{cases}$$

Here h_t is the symbol for the output of the current decoder and $\bar{h}_s$ is the symbol for another

output taken from the input encoder. This one is the entire set of encoder states. Inkawhich et al (2018)[2] uses the 'dot' variety.

The formula is used after the decoder Gated Recurrent Unit calculates it's hidden state. It is below.

$$\boldsymbol{score = h_t^{\intercal} \bar{h}_s}$$

Basically the output of the current decoder is transposed. Then it is multiplied by the hidden value from the entire set of encoder states. The result is multiplied by the Gated Recurrent Unit decoder output, and then passed through a 'tanh' activation layer.

1.6 Sequence to Sequence Chatbot

Vinyals et al (2015)[1] make an interesting proposition. They say that it's possible to make what they call a Neural Conversational Model by constructing a Sequence to Sequence model, but instead of using two corpus from different languages a single language is used for both source and target.

Chatbots have for a long time been constructed using AIML. AIML, (Artificial Intelligence Markup Language) requires hand coding individual rules for every input and response. A Neural Conversational Model would not use those sorts of rules.

To create a model like this more is required than just a single input and output language. There must be a relationship between the source and the target. We want there to be a question-and-answer-like relation. Finding a corpus like this can be difficult.

It would be easy to duplicate the input source material in the target corpus. This would produce auto-encoding. The model would learn to repeat everything that was given to it in it's output. Though the model learns a task, it is not the dynamic task we want. Conversations, on the other hand, supply the relationship we are looking for. Starting with almost any question sentence in a conversation, the sentence following it answers the question posed. The latter fills a space that is opened up by the former.

What would be a good candidate for this kind of verbal play? Vinyals et al (2015)[1] use a movie transcript data set. Essentially they take movie dialogue and break it into sentences. Even numbered sentence are the source material, and odd numbered sentences are the target. Using this

method there are times when the source and target are not from the same conversation, like times in a movie dialog when the scene switches from one locale to another. Comparing this, though, to the number of times that the two sentences are from the same dialogue, the movie transcript database serves very well.

Training for this model is relatively straight forward. There is a problem though. We can follow the loss and make sure it is decreasing, but the accuracy tells us nothing and in this case does not increase as we would like it to. The loss goes down but the accuracy does not go up.

Figure 1.4: Loss and Accuracy: Red is accuracy and blue is loss.

This is because the source and target do not have the same meaning. The model does learn the task at hand but during training we have to ignore the accuracy. Success is usually measured by the accuracy of the holdout test set. Here we must measure the success with a subjective examination of the trained model.

We have to interactively give the model questions that we might ask someone that we are having a conversation with, and then see how it answers. Over and over we have to test the model. If we are satisfied with the answers then the training was a success.

Chapter 2

Transformers and The Generative Pre-training Transformer 2

2.1 Transformer and Attention

The Transformer is a mechanism that is based entirely on attention. Strictly speaking this is not the attention explored in the Sequence to Sequence model in Chapter 1, though there are some similarities. It is a model that uses no recurrent components and no convolution components.

Recurrent components have some negative qualities. Recurrent components are hard to run with batch input data. Also they do not work with very long data strings.

If a batch of data is to be cycled through a recurrent component, all that data has to go through the component at the first cardinal position before it is cycled through the second or third position. In the first time step everything needs to go through the first Recurrent Neural Network module. Then the data can be moved along to the second Recurrent Neural Network module in the second time step.

Also because the Recurrent Neural Network is so heavy with Neural Network components, many weights and biases, though they can remember patterns, they loose some information with every pass. This is why there is a practical limit to the length of the input sequences that the typical Recurrent Neural Network can use. This is why the length of sentences in network models that use the Recurrent Neural Network are short.

Transformers use no Recurrent Neural Network components. Their operations can be parallelized so that large batches of data can be processed at once during the same time step.

Longer sequences can be considered as well, so Transformer input can contain longer English language sentences and even paragraphs.

Byte Pair Encoding

BPE stands for 'Byte Pair Encoding.' WordPiece is a particular implementation of Byte Pair Encoding.

WordPiece is used by some transformer systems to encode words much the way that Word2Vec does. Like Word2Vec, WordPiece has a vocabulary list and a table of embeddings that maps one word or token to a vector of a given size.

WordPiece, though, handles Out Of Vocabulary (OOV) words gracefully. It breaks large words into smaller pieces that are in the vocabulary, and has a special notation so that these parts can easily be recombined in order to create the input word again. Byte Pair Encoding is not so interested

in pre-trained word embeddings like Word2Vec and Glove.

For the Generative Pre-training Transformer 2 a version of Byte Pair Encoding is used instead of a vocabulary system like Word2Vec or Glove.

Attention

Attention mechanisms are used in a similar way in three places in the model. The first implementation of the Self Attention is discussed below.

It should be noted that input to the Transformer is strings of words from a desired input language. Output is similarly words in a given language. Input words are treated very much the way that they are in Sequence to Sequence models. A word is translated to a number and that number indexes an entry in a word-to-vector table. From that time forward a word is represented by a vector of floating point numbers. In a transformer this word vector can be large. In the original paper Vaswani et al (2017)[3] use a vector size of 512.

Scaled Dot-Product Attention

The Transformer has a signature self-attention mechanism. This is possibly one third of the entire Transformer mechanism, but a variety shows up in the other two-thirds.

The first thing that happens is the input word vector is converted to three other values. These new vectors are like the input vector but they have a smaller dimensionality. Converting the word vector in this way is accomplished by three simple matrix multiplication operations.

In the diagram below a simple conversion of this type is illustrated. In the diagram we convert a vector with dimension of 1x3 to a dimension of 1x2. In reality we are converting a vector from 1x512 to 1x64. This is a division of 8.

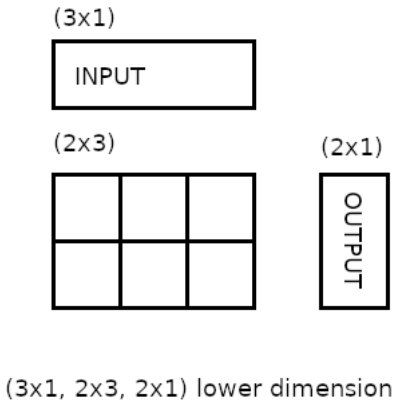

Figure 2.1: Lowering Dimensionality

One thing we want to do is to preserve the dimension of our vector. We start with a 512 sized floating point vector and after some processing we want to return to the same size. Before that is done the vector is processed at the smaller size of 64.

In this self-attention scheme three vectors are actually required. All three vectors are sized 64, and all three are converted by separate matrix multiplication operations. The weights to convert each of the three vectors are different. For this reason the new smaller vectors are all different.

The smaller vectors individually are called q, k, and v. They can also be referred to as larger matrices. The new vector matrices are denoted as Q, K, and V. Q stands for 'Query'. K stands for 'Key'. V stands for 'Value'.

The Query value is multiplied by the Key values from all vectors in the input. This multiplication is 'dot-product' multiplication. When it is done all keys will have low output except those that are closest to the Query. Then the results are passed through a softmax function. When this is complete there will be a single vector that is close to 1 and another group of vectors that are all close to 0.

The vector produced by multiplying the softmax with the V values of every word produces a single word vector that is close to its original value, and many others that are near zero.

This formula from Vaswani et al (2017)[3] shows the process.

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

Here the value of $\sqrt{d_k}$ is used to limit the size of the QK^T output and d_k is the dimension 512. Without this the softmax function has to deal with much larger numbers. Smaller numbers for the softmax are preferred.

The function can actually perform this on large matrices with high dimensionality, in parallel. This parallel matrix operation increases the speed of training.

In the green triangle in the Figure 2.2 we preform the multiplication and selection that was just described.

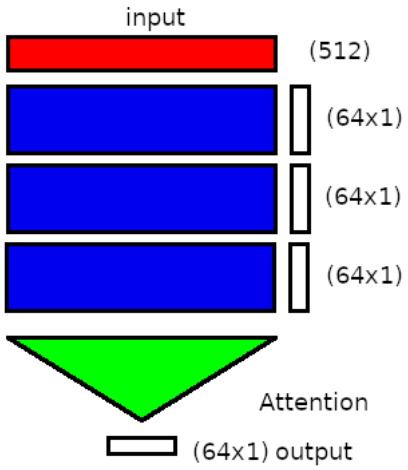

Figure 2.2: Attention Output

Finally the output we calculated above must be returned somehow to the input dimensionality. This is accomplished by duplicating the procedure described eight times with eight separate weights. When this is done the output of the attention mechanism is concatinated together, returning the output to the proper size.

This multi-headed approach allows different heads to learn different types of relationships, and then when they are summed together the learned relations are recovered and contribute to the output.

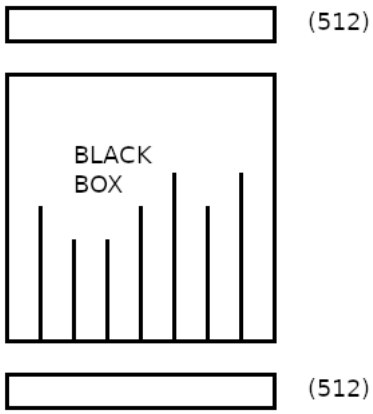

Figure 2.3: Matching Input and Output

Later the output is passed through a feed forward network. After that it is combined with the original input again through addition. Then the vectors are normalized. This makes sure that the values are all within reasonable ranges.

This describes the encoder section. There are two other attention segments. Together these two sections combine to form the decoder section.

Transformer - Decoder

The decoder is composed of two attention mechanisms and a feed-forward segment. The result of the encoder's work is passed to the decoder and remains applied to one of the decoder's attention mechanisms throughout the decoder's work. In that one of the two attention mechanisms of the decoder the 'Key' and 'Value' matrices are imported from the encoder.

While the encoder takes in the entire input and attends to whatever portion of that input it finds to be important, the decoder is interested in producing one token at a time.

During inference it produces a token and then it adds to that token, one at a time, until the decoding is finished and something like a sentence is produced. It can attend to any part of the output it has already produced. During training the decoder is exposed to the target sequence under a mask. The mask prohibits the decoder from seeing parts of the target that it should not. This mimics the inference setup and still allows for large input matrices.

We illustrated in the Sequence-to-sequence discussion the importance of the single 'thought

vector'. The Transformer can be seen as having a thought-vector also. There is a corridor of data from encoder to decoder. Two important smaller vector-sized inputs from the encoder are ultimately required in the decoder. They represent the 'Key' and 'Value' matrices. The matrices required are the size of the smaller, reduced, vector. The full sized vector is transported from the encoder and is reduced dimensionally in the decoder to two smaller matrices.

There is also a second attention mechanism in the decoder. It works solely on data from the decoder itself. It works very much like the attention mechanism from the encoder - only it attends to every word of output as opposed to the entire input sequence. It passes it's output to the layer described above. This data is lowered in dimensionality and becomes the 'Query' matrix for that layer.

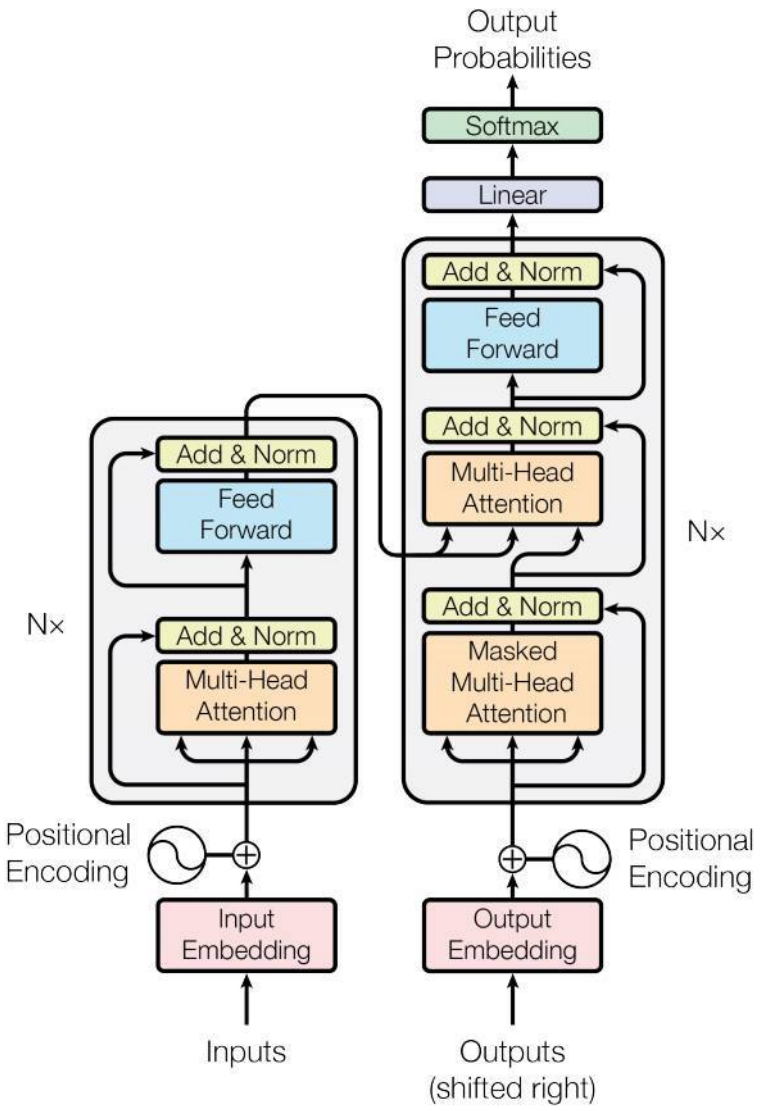

Figure 2.4: Transformer Encoder and Decoder - Vaswani et al(2017)[3]

Transformer - General

The input of the Transformer encoder and decoder employ not only a word-vector table, but also a positional encoding scheme. The model adds to the input vector information that the it can then use to learn the position of words in a sentence.

Words that are early in the sentence have a certain appearance and words later on appear

differently. The Encoder and Decoder use sine and cosine waves to impart this information onto the sentence sequence.

Meanwhile at the output of the decoder the output vectors are processed through a linear matrix which increases the vector's dimensionality so that the output vector is the size of the output vocabulary dimensionality. After the linear matrix the vector is processed by a softmax function. Then the highest floating point value in the new larger vector is the index of the chosen output word.

The Transformer model takes large memory resources, large corpus resources, and large training time resources. Without these components the Transformer is not suitable for many projects.

Pre-Training

Pre-Training is when the authors of a model train an instance and then make the model available to the public on-line. This is helpful for the average programmer interested in Neural Networks. Training an instance of the transformer model can use up computation resources for days, and require hardware that is costly. Usually the cost of producing a trained model is prohibitively expensive.

After acquiring a trained model, the programmer goes on to adjust the model to their task. Adjusting a pre-trained model to a given task is called 'Transfer Learning'. Many tasks lend themselves to Transfer Learning. Conceptually a model can be fine-tuned to any problem and many problems can be addressed with good results after only modest fine-tuning.

There is a pre-trained Transformer model called BERT. BERT stands for 'Bidirectional Encoder Representations from Transformers'. The BERT files available on-line are mainly for classification tasks. They allow for input that uses a vocabulary size that is very large, but the output is meant to be smaller.

The 'Bidirectional Encoder Representations from Transformer' models use a bidirectional training method. During training the BERT model might be presented with a sentence with a word missing. The training task is to fill in that blank. The model can look at the sentence from any direction in order to find that word. This is a very simplistic explanation for what is termed a Masked Language Model.

Visualization - Transformer

In order to visualize what is happening during inference we have colorful charts that we can look at. In this chart we are looking at how each word attends to all the other words in the input text.

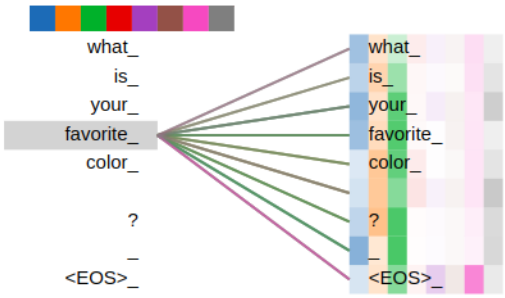

Figure 2.5: Visualized Attention – 'favorite' shows attention to most words in the sentence.

Significant, but not shown, is that other words, like 'what' and 'your', do not have strong attention to words in the text. In a chart like this one they would show no colors on the right and no lines connecting the left to the right.

This diagram is from the Transformer with the larger hyper-parameter set that we describe in Chapter 3, trained on the movie dialog corpus.

2.2 The Generative Pre-training Transformer 2 Model

'Generative Pre-training Transformer 2' (GPT2) is a large model. It is based on the Transformer from Vaswani et al (2017)[3] but there are some major changes. The model uses the encoder portion of the Transformer without the decoder. There are some other changes to the output layers. The biggest difference is that it's pre-trained and downloadable.

It still uses Scaled Dot-Product Attention. A model diagram is taken from Radford et al (2018)[12].

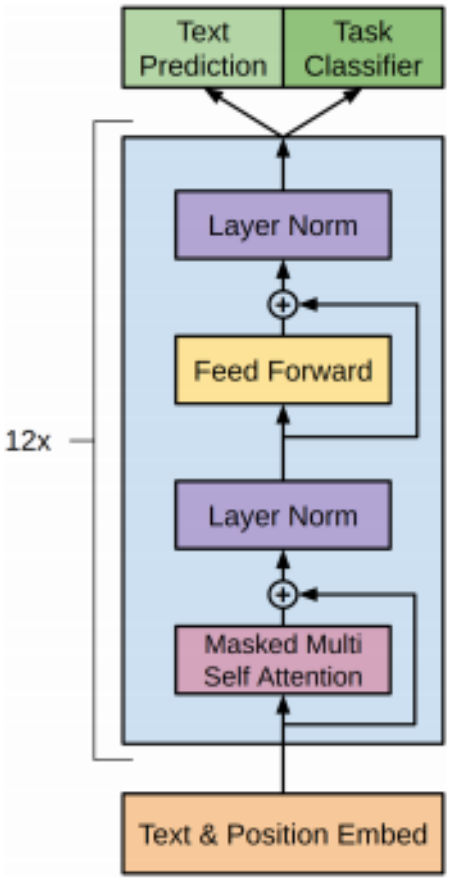

Figure 2.6: GPT2 - Radford et al(2018)[12]

There are several sizes of pre-trained 'Generative Pre-training Transformer 2' model. They are all rather large. The smallest model matches the size of the largest 'Bidirectional Encoder Representations from Transformers' model. The GPT2 models input and output text sequences. In this way they are preferred for our application over the BERT models.

The 'Generative Pre-training Transformer 2' models are trained on a corpus called WebText. WebText is a 40GB corpus that is taken from the Reddit web site. All the material comes from before 2017 and all the material has a 'carma' rating of 3 or better. 'Carma' is a rating system used internally on Reddit.

In their paper Radford et al (2019)[4] show that their model can generate text from a seed sentence or paragraph. At the time the case was made that the largest 'Generative Pre-training

Transformer 2' models should not be released because of their ability to generate text that might fool humans into believing that another person was responsible for the text. Later the larger models were released to the public.

Size	Parameters	Layers	d_{model}
small	117M	12	768
medium	355M	24	1024
large	774M	36	1280
x-large	1.5B	48	1600
xx-large	8.3B	72	3072

At the time that the first 'Generative Pre-training Transformer 2' model was released the size of the models was mis-stated, but the documentation was not updated immediately. Most values in the table above show sizes that were actually released. The final xx-large model was trained by NVIDIA Applied Deep Learning Research (2019)[13] and was not released to the public.

The 'Generative Pre-training Transformer 2' models also work in many circumstances in 'zero-shot' mode. This is when you use the model pre-trained but without transfer learning. There is no extra training that goes on to make the model suit the task. It is used 'as is'.

For the chatbot the model with 117 million parameters worked. Some programming was required to make the model output look like chatbot output, but the model itself was not modified.

Later when the larger 774M model was released it was used as a substitution for the 117M model. The test worked, and returned answers that were more well formed than the small model. The larger model does not fit on a Raspberry Pi and so it was not employed on a permanent basis. Using the extra large model in a chatbot was not attempted.

Application Details

The model is described in Radford et al (2019)[4] and the accompanying blog post. The model is trained on English without a stated problem. Large neural network models are usually trained with a stated problem in mind. Rather famously this model is used after training to generate English language text. The model takes input from the user, a premise or summary of what is to be generated. The model also takes as input a number called the 'temperature.' Then the model generates output. As the 'temperature' is set higher the output is more fanciful. There is also a tune-able parameter for the output length.

Given the ability of the model to invent content, it was determined by the authors that the 'large' model should not be released to the public at first. Months later the 'large' model was released.

For our chatbot we set the temperature to a low number. We set the length of the output to a sentence-length number of tokens. Then as input we use the output from the speech-to-text translator.

The output is interesting but not useful right away. Heuristics are employed to clean the output and render a short single sentence. This is our final output.

Because the input is meant to be a number of sentences, and because we are using a transformer-based architecture, we have room in the input string to add more information along side the user's question. In this respect the model acts to summarize the input.

With every input string we include a set of three or four sentences. They include the time, the bot's name, and the bot location and occupation. All of these are invented. What happens is the chatbot summarizes the input and only if the information is relevant then the same information is used by the model as output. Making this possible is the fact that a transformer can accept much longer input strings than a Gated Recurrent Unit, and generate much longer output strings.

Visualization - GPT2

During inference the Scaled Dot Product Attention in the GPT2 focuses on certain words as it processes input text. Here the word 'favorite' shows a relationship to many of the other words in the text.

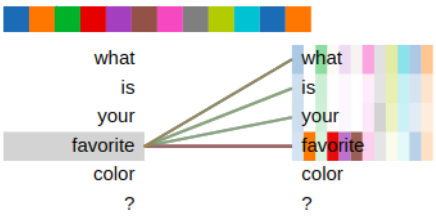

Figure 2.7: Visualized Attention GPT2 – 'favorite' shows attention to most but not all words in the sentence.

In our experiments the phrase 'What is your favorite color?' is answered with 'I love the colors of the rainbow.' This answer does not mention a specific color, as one might expect it should. The diagram might support this observation.

Chapter 3

Experiments

3.1 Approach to the Study

Several tasks are necessary for the project. One task is to implement the algorithms for different models, one the sequence to sequence model and the other the transformer model for a generative chatbot and finally the Generative Pre-training Transformer 2 model.

We will not try to rewrite the transformer or Generative Pre-training Transformer 2 model ourselves.

In this project we attempt to load as much of out chatbot code onto a Raspberry Pi as possible. We have trained models using the pytorch and tensorflow libraries. These models are responsible for taking in English sentences and producing English output. There is another part of the typical Raspberry Pi setup that includes another neural network component. Speech to text models, which our application requires, rely on large neural network resources. For this purpose we use speech to text resources supplied by Google on the google cloud. To include speech to text libraries locally on the Raspberry Pi would be too costly in computation time and resources like RAM. It would also be complicated to implement technically. It could easily comprise an entire project on its own.

Unfortunately the speech to text resources supplied by Google cost money. To use the service you need to have a billing account with Google.

The speech to text service used on the project and the memory limitations on the Raspberry Pi leads one to ask the question weather the neural network responsible for the chatbot function could not be servable from some faster machine located somewhere on the internet. At this time we are not interested in serving these resources. It would entail two calls from the Raspberry Pi for every sentence. This complicates things and also has a time overhead.

Also, we have several models that we want to test. To test them all would require several servers. Also we use both Pytorch and tensorflow. Tensorflow has 'tensorflow-model-server' for serving models, but Pytorch has no equivalent.

It is important to note that the large Generative Pre-training Transformer 2 model specifically could be served from a remote computer and it would operate faster. Currently on the Raspberry Pi decoding a single sentence takes approximately 13 seconds. Even so, we prefer to install our trained models on the Raspberry Pi directly.

Model Name	File Size	RAM Train	RAM Interactive	Pretrained Weights	Hand Coded
Seq-2-Seq/Tutorial	230 M	1.1 G	324 M	NO	NO
Transformer/Persona	25 M	556 M	360 M	NO	PARTIAL
Transformer/Movie	550 M	6.5 G	1.5 G	NO	PARTIAL
GPT2 small*	523 M	5 G	1.5 G	YES	NO

* a large GPT2 model exists, but it is not small enough to fit on a raspberry pi.

3.2 Model Overview

Here we itemize a short description for each row in the table.

- **Sequence to Sequence - Tutorial** This model uses the sequence to sequence architecture and the Gated Recurrent Unit component. We hand-coded our own example of this model but it performed poorly. This model is the slightly modified version of the Sequence to Sequence model based on the tutorial from Inkawhich et al (2018)[2]. It actually uses the Movie Dialog corpus.

- **Transformer - Persona** This model uses a Tensorflow Transformer architecture. There was some coding involved to get the model to interface with the text-to-speech and speech-to-text libraries. There was also some coding to load our own corpus data during training. The model parameters describe a rather small model. This model also uses the Persona Dialog corpus.

- **Transformer - Movie** This model is based on the transformer model above but uses the Movie Dialog corpus and a parameter set that is larger. In many ways this model is bigger than the model that uses the Transformer and the Persona corpus.

- **GPT2 small** This model was downloaded from the internet. It fits on a Raspberry Pi 4B with the 4GB RAM option. Some modification was made so that model output was suitable for our purposes.

3.3 Setup

We use linux computers, sometimes with GPU hardware for parallel processing. We also use the Python programming language. Code from this project can be run with the 3.x version of Python.

When the project was started we did some programming with Keras using Tensorflow as a back-end. Keras was later discarded in favor of Pytorch. Pytorch as a library is still under development at the time of this writing.

Some of the Generative Pre-training Transformer 2 code uses Pytorch. Some of the Transformer and Generative Pre-training Transformer 2 code uses Tensorflow. There is a repository on Github that has the GPT2 trained model using Pytorch instead of Tensorflow.

We use github as a code repository. Code corresponding with this paper can be found at: https://github.com/radiodee1/awesome-chatbot .

As a coding experiment we rewrite the code for the sequence-to-sequence Gated Recurrent Unit model. We have varying amounts of success with these experiments. We do not rewrite the Generative Pre-training Transformer 2 code from the Tensorflow or Pytorch repository.

3.4 Graphical Processing Unit vs. Central Processing Unit

A CPU has a number of cores, a number usually between 2 and 16. A CPU is designed, though, to execute one command at a time. This allows for a logical chain of actions that can be programmed for execution. A CPU has limitations when it comes to executing matrix multiplication. Matrix multiplication using a CPU can take a long time.

GPUs, Graphical Processing Units, have the ability to address tasks like matrix multiplication with many more processing units at once. The GPU speeds up parallel processing and have a benefit to neural networking training tasks that the CPU doesn't have.

Unfortunately state of the art neural network models are larger than the capacity of a single GPU. Some models are trained on many GPUs simultaneously. It is not uncommon for a model to train on a computer with eight GPU cards for many days. Training these models is prohibitively expensive for the average programmer. It is possible to rent time on Amazon Web Services or Google cloud with well outfitted computers but this can be costly.

This sort of situation is addressed partially by the Transfer Learning scheme. In Transfer Learning someone else trains the model and makes the trained version accessable to the public. Then the average programmer downloads the model and fine tunes it to their task.

This would be fine if there were a model for every task. Though many models exist there seems to be many tasks that are not adressed. It seems that there is always the opportunity to train a

larger model by utilizing the CPU and training for long periods of time. This arrangement is not advised for the sort of experimentation where it is not a certainty that the output will be successfull. If the goal is to do somehting that might not work, don't undertake it or use a Amazon or Google computer. If success is assured and time is plentiful continue with the CPU.

In this paper the GRU based Sequence-to-sequence model and the Tensorflow based Transformer model were trained from scratch on a CPU laptop. In the case of the Transformer, several days were required for training.

3.5 Raspberry Pi

A Raspberry Pi is a small single board computer with an 'arm' processor. There are several versions on the market, the most recent of which sports built-in wifi and on-board graphics and sound. The memory for a Raspberry Pi 3B computer is 1Gig of RAM. Recently available, the Raspberry Pi 4B computer can sport 4Gig of RAM.

It has always been the intention that at some time some chatbot of those examined will be seen as superior and will be installed and operated on a Raspberry Pi computer. If more than one model is available then possibly several models could be installed on Pi computers.

For this to work several resources need to be made available. Pytorch needs to be compiled for the Pi. Speech Recognition (SR) and Text To Speech (TTS) need to work on the Pi.

For one of the transformer models to work Tensorflow needs to work on the Pi.

All the files that are trained in the chosen model need to be small enough in terms of their file size to fit on the Pi. Also it must be determined that the memory footprint of the running model is small enough to run on the Pi.

In the github repository files and scripts for the Raspberry Pi are to be found in the 'bot' folder.

Early tests using Google's SR and TTS services show that the Pi can support that type of functionality.

Google's SR service costs money to operate. Details for setting up Google's SR and TTS functions is beyond the scope of this document. Some info about setting this up can be found in the README file of this project's github repository.

The pytorch model that is chosen as best will be trained on the desktop computer and then the saved weights and biases will be transferred to the Raspberry Pi platform. The Pi will not need to

do any training, only inference.

3.6 Tensorflow vs. Pytorch

Tensorflow is a Google library. Pytorch has it's roots with Facebook. Both run in a Python environment. The learning curve for Tensorflow is steeper than for Pytorch. Pytorch offers the programmer python objects that can be combined to create a neural network. Tensorflow has different pieces that can be combined, but they cannot be examined as easily at run time.

Tensorflow has a placeholder concept for inputting data and getting back results. You set up these placeholders at design time. They are the only way of accessing your data at run time.

Pytorch objects interact with Python more naturally. You can use print statements in your code to show data streaming from one object to another. This is possible at run time.

In favor of Tensorflow, it has a good tool for visualization which can print out all kinds of graphs of your data while your model trains. It is called Tensorboard.

3.7 Speech and Speech To Text

Google has python packages that translate text to speech and speech to text. In the case of text to speech the library is called 'gTTS'. In the case of speech to text the library is called 'google-cloud-speech'.

The gTTS package is simple to use and can be run locally without connection to the internet. The google-cloud-speech package uses a google cloud server to take input from the microphone and return text. For this reason it requires an internet connection and an account with Google that enables Google cloud api use. Google charges the user a small amount for every word that they translate into text.

Both of these resources, the text-to-speech and speech-to-text, work out of the box on the Raspberry Pi, but configuring speech-to-text for the Pi is not trivial. The user must register a billing account with Google cloud services. In return for this registration the user is allowed to download a json authentication file. The file must be copied to the Raspberry Pi.

Furthermore an environment variable must be set that points to the authentication file. The variable is called 'GOOGLE_APPLICATION_CREDENTIALS'. This environment variable has to

be set up before the respective model runs. When the model is launched on startup it may not be launched as a regular user. The model may be launched as, for example, the root user. Somehow the environment variable must be set along with the launching of the neural network model.

The operating system on the Raspberry Pi is based on Debian Linux. In this operating system there is a file which is run immediately after the basic system starts up. This script is called '**/etc/rc.local**'. It is sufficient to put the environment variable there and follow it with the launching of the model. To ensure that the process goes without a hitch, we attempt to combine the setting of the environment variable with the launching of the program in a single line of code.

3.8 Corpus Considerations

We have collected several data sets for the training of a chatbot model. Firstly we have a corpus of movie subtitles. Secondly we have a 'JSON' dump from Reddit that is downloadable. This is not the same Reddit data that the authors of GPT2 use. Finally we have the corpus described by Mazaré et al(2018)[10]. This final corpus is designed for training the chatbot task specifically. This is referred to as the Persona corpus.

The movie corpus is medium sized and the Reddit 'JSON' download is large and filled with hyperlinks and sentence fragments.

At the time of this writing we are using the movie subtitles corpus and the Persona corpus. We use the movie corpus because it is smaller. Both the movie corpus and the Reddit corpus are described as noise filled, so it is likely that neither one is perfect for the training. The movie corpus is easier to deal with if we are training on a single processor. In the future if we can train in a faster environment the Reddit corpus might be superior.

For the Persona corpus the text is organized into 'JSON' objects. There are several different repeated labels. Some of the text is meant to be used in question and answer pairs. There is also some very specific information there that is not organized in this kind of pattern. When we take apart the Persona corpus we find that the sentences labeled with the 'history' tag are most suited to our task. We record these values only and discard other labels.

3.9 ARMv7 Build/Compile

Pytorch 'torch' Library 1.1.0 For ARMv7

We compile the Pytorch library for Raspberry Pi. We use several virtualization techniques to do this compilation. The result of those efforts is a Pytorch python 3.7 library for the Raspberry Pi.

On their web site Milosevic et al (2019)[14] compile Pytorch 1.1.0 for the Raspberry Pi. We follow their instructions closely. We are able to build the package for the ARMv7 platform.

The instructions called for constructing a change-root environment where a Fedora Core 30 linux system was set up. Then the ARMv7 system was used in the change-root environment to compile the Pytorch library for the 1.1.0 version.

The production laptop used for development ran Ubuntu linux. For this reason a Virtualbox emulation was set up with Fedora Core 30 on it. Inside that emulator the change-root environment was set up. The library was compiled there successfully.

There are two problems with the resulting built python package. Firstly there is an error in python when importing the torch library. The error reads 'ImportError: No module named _C'.

After some research it is clear that the build process for ARMv7 creates some shared object files that are misnamed. A hackey fix is to find the misnamed files and make copies of them with a suitable name. The same outcome could be assured by making symbolic links to the misnamed files with proper names.

There are three files misnamed. They can be found at '**/usr/lib/python3.7/site-packages/torch/**'. They are named with the same convention. They all have the ending '.cpython-37m-arm7hf-linux-gnu.so'. We want to rename them with the much shorter '.so'. The files are then named '**_C.so**', '**_dl.so**', and '**_thnn.so**'.

This takes care of the 'ImportError'. The second problem is that the version of GLIBC in the change-root environment does not match the GLIBC library in the Raspberry Pi Raspbian distribution. This produces the following error: '**ImportError: /usr/lib/x86_64-linux-gnu/libstdc++.so.6: version 'GLIBCXX_3.4.26' not found**'.

This is solved by rebuilding the package with Fedora Core 29 instead of 30.

Pytorch 'torch' Library 1.4.0 For ARMv7

We recompile the Pytorch library for the Raspberry Pi. We use debian virtualization techniques for the compilation. Because ubuntu is a debian derivative it is not necessary to run the process in a Virtualbox container.

In addition to this, the files created by the compilation are properly named. There is no need to go to the directory '**/usr/lib/python3.7/site-packages/torch/**' to change anything.

The time spent compiling the software is approximately 5 hours. Time spent with the Virtualbox container was easily twice that. The time spent on the Raspberry Pi executing a single Generative Pre-training Transformer 2 question and answer remains about 13 seconds, so there was no gain in that respect.

There were several small hurdles to completing the compilation. Firstly the 'debootstrap' command needed to be employed at the start. Debian Stretch was used as the host operating system. It was felt that if it was used that the GLIBC compatibility problem would not be faced. This turned out to be the case.

There are some dependencies that need to be installed on the 'chroot' environment for Pytorch to compile. One of these is that is important is 'libblas3.'

Then Python 3.7 needed to be built on Stretch. The Stretch program repositories use Python 2.7 and 3.5 . The Rapbian operating system on the Raspberry Pi 4B is based on Debian Buster and uses Python 3.7. After compiling Python 3.7 the Git program needed to be compiled from scratch. Git on Stretch has a issue that is fixed upstream, but we want to use Stretch because of the GLIBC issue. Instead of using the upstream fix, we compile Git ourselves.

It is conceivable that the GLIBC issue would not be important if the 'chroot' environment used Debian Buster, since that is the basis for the current Raspbian operating system. The Stretch operating system solution works though.

Finally the Pytorch program needed to be built. We disable CUDA and distributed computing as neither exists on the Raspberry Pi.

Docker Container 'tensorflow-model-server' For ARMv7

The Google machine learning library for python uses a standalone program called 'tensorflow-model-server' for serving all tensorflow models in a standard way. The program has not been officially

compiled for ARMv7. There exists, though, a docker image that will run on ARMv7.

Docker can be run on the Raspberry Pi in the ARM environment. Below is a terminal excerpt that shows how to do this.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ curl -fsSL test.docker.com -o get-docker.sh
$ sh get-docker.sh
$ sudo usermod -aG docker $USER
```

After the last command you need to log out and then log in again to take advantage of the newly installed docker.

The original idea was to follow someone else's instructions and compile the Docker Container for the ARMv7. Then the executable would be removed from the container and used natively in the Raspberry Pi.

It was found that there existed a version of the Docker Container Daemon that ran on the Raspberry Pi. All that remained was to write a Docker Container script that interacted with the existing ARMv7 container. The author of the original container is Erik Maciejewski (2020)[15].

'Tensorflow-model-server' is used on the localhost internet address, 127.0.0.1, with a port of 8500. tensorflow-model-server is meant for serving neural network resources on the internet, but with careful planning it works on the Raspberry Pi.

3.10 Experiments - Installations

In the section above we describe the workings of a transformer and the workings of Generative Pre-training Transformer 2. We propose they are similar. Here we distinguish between the two. For the experiments section they are totally separate.

We have several basic neural network models. One is the basic sequence to sequence model typically used for neural machine translation. We also have the transformer and the Generative Pre-training Transformer 2. We try to touch on each model type and we also distinguish between chatbot operation and smart-speaker operation. This gives us six sections.

The Generative Pre-training Transformer 2 code is versatile. We use it for the chatbot problem.

The model is pretrained. We experiment with transfer learning and further training of the Generative Pre-training Transformer 2 model, but in our case it does not improve the model's performance.

When we talk about the chatbot problem on Generative Pre-training Transformer 2 we are talking about a PyTorch version of the GPT2 code.

We found that the chatbot with the hand coded sequence to sequence Neural Machine Translation did not work very well. There is a tensorflow transformer-based model that worked marginally well. We found that the Generative Pre-training Transformer 2 chatbot worked well enough with the 'zero-shot' setup so that fine-tuning was not necessary.

Fine tuning on the Generative Pre-training Transformer 2 problem actually had a negative effect. To fine-tune the model also involved training the model on Tensorflow and then translating the model to PyTorch when that was done.

We use speech recognition and speech to text libraries to allow a user to give the chatbot model auditory input.

We also train the transformer to do the chatbot task. This works better than the the hand coded sequence to sequence model but not as well as the Generative Pre-training Transformer 2.

For the sake of experimentation we use a sequence to sequence Gated Recurrent Unit tutorial and implement the chatbot. This model works well and allows us to compare that model with the Generative Pre-training Transformer 2 model.

Finally we describe installing the chatbot in small computing platforms, the Raspberry Pi 3B and 4B, in an attempt to create a smart speaker.

All of the three mentioned models work on a laptop computer. For the laptop we experimented with a setup where the model would launch applications for the user when the user asked the model to. These audio cues were interesting but were in no way the focus of our experiments.

It should be noted that at some point we would like to describe some of our subjective results with the different models. It is difficult to measure our results objectively because we are using code most closely associated with a translation task.

If we were measuring progress with an actual translation task accuracy could be measured as a score that improves when the output matches the meaning of the input, albeit in a different language. We don't use an actual translation task, we use something close to one. The problem is that though the input and the output are in the same language, the input and output have different meanings. We do not dictate what the correct output would be for a given input. The output just has to make

sense as a reply in the English language. We would actually prefer that the output not have the same meaning as the input. This makes it difficult to calculate an objective accuracy score. This is touched on in the paper from Vinyals et al(2015)[1].

Loss can still be monitored during training. When the loss stops decreasing you know you should stop training as the model is probably overfitting.

Questions

Below is a list of questions asked of all models.

```
Hello.
What is your name?
What time is it?
What do you do?
What is your favorite color?
Do you like red?
Do you like blue?
What is your favorite candy?
Do you like ice cream?
Good bye.
```

For comparison there are four models. Subjectively the first transformer model did not perform as well as the Generative Pre-training Transformer 2 model. It did not perform better than the Gated Recurrent Unit model either. The model from the Gated Recurrent Unit tutorial performed well. It was better than the initial Transformer model and on par with the larger Transformer model. It was not better than the Generative Pre-training Transformer 2.

3.10.1 Chatbot - Gated Recurrent Unit Model

We have two models based on the sequnce to sequence architecture proposed by Vinyals et al (2015)[1]. One model was largely written by hand. This first model does not work very well. The second model was taken largely from an online tutorial by Matthew Inkawhich (2018)[2].

We trained the first sequence to sequence model on a large english corpus in an attempt to produce a chatbot.

For this sequence to sequence model we want to use text found in a movie dialog corpus (Danescu-Niculescu-Mizil et al, 2011)[16].

In our experience with coding our own chatbot we find that the model learns a single English sentence that can be used for most replies.

Subjectively the Gated Recurrent Unit chatbot that we code is unusable.

For example our chatbot frequently replies to input with the phrase 'I don't know'. This is the most common output from our model. In this respect our original chatbot responds poorly.

A large portion of the time the bot's response to a question is the same. This is interesting in that the chatbot has identified that this phrase is a suitable answer to many questions, but it is disappointing that there is not more variety to the output.

Occasionally the sequence-to-sequence model will reply with the phrase 'I'm sorry.' This happens very infrequently and it is not clear why the model chooses to reply this way.

The second sequence to sequence model performed exactly as expected. It answered a number of subjective questions with original, authentic answers. The answers all made sense in the English language.

Questions

This is the sample question list as it was answered by the better model.

```
> hello .
you re a good man .
> what is your name ?
smith .
> what time is it ?
i don t know .
> what do you do ?
i m a comedian .
> what is your favorite color ?
i don t know .
> do you like red ?
no .
```

```
> do you like blue ?
no .
> what is your favorite candy ?
it s a big one .
> do you like ice cream ?
no .
> good bye .
good bye .
```

Checklist:

- ☑ All the responses are in plain English. There is no gibberish.
- ☑ There is a variety of answers. Not all answers are the same.
- ☐ It is debatable weather or not the answers to the questions about 'favorite color' and 'favorite candy' are good. The model could have a set of easy answers that it can use for this kind of question.
- ☒ 'No' is a safe answer for many types of question as it is clearly English, it follows logically, and it is short and easy to remember. Another safe answer is 'I don't know'. This model uses that answer at times.
- ☑ The model answers well to 'Hello' and 'Good bye'.

3.10.2 Smart Speaker - Gated Recurrent Unit Model

The Gated Recurrent Unit model was installed on a Raspberry Pi. This allowed us to test out speech-to-text and text-to-speech libraries. The Raspberry Pi model was 3B. The RAM requirements were less than 500MB and the trained model answered questions on the Raspberry Pi almost instantaneously.

For this experiment we compiled the Pytorch library for Raspberry Pi.

The Raspberry Pi was outfitted with a microphone and a speaker and nothing more. It was also configured so that the Pytorch sequence to sequence model ran automatically on startup.

The model requires access to the internet for the exchange that the speech to text software has to make with the Google servers. If there is no internet the model doesn't work.

As there was no monitor and it took some time for the model to launch, the program was coded to beep when the model was ready to accept input. At that point the model beeps once.

3.10.3 Chatbot - Transformer Model with Persona Corpus

Using the Persona corpus we trained a transformer model to use as a chatbot. This transformer was not pre-trained with any large corpus, so this example did not use transfer learning. The Persona corpus comes from Mazaré et al(2018)[10].

This model uses the tensorflow library, not Pytorch, and a transformer model that is somewhat small.

The memory footprint of the model while it was running was below 1 Gigabyte. It is conceivable that the model could be installed on a Raspberry Pi board but it requires a python package called 'tensorflow-model-server' and this package would have to be built from source for the Raspberry Pi.

Training of the model followed a certain pattern. First the model was trained on the persona corpus until a familiar pattern emerged. When the model began to answer all questions with the phrase "I don't know" training was stopped.

At that time the corpus was modified to include no sentences that have the word "don't" in them. Training was started again until the output contained nothing but the phrase "I'm sorry."

At that time the corpus was modified to include no sentences that have the word "sorry" in them. Training was started again and was continued for some period. Training was stopped. A further segment of training was not attempted.

At this point, after looking at the change in loss, further training was not thought of as helpful. Loss stopped improving at some point in this process, and this lack of improvement was taken as a sign that further progress was not likely.

Subjectively the transformer model is better than the original Gated Recurrent Unit model. It is not better than the Gated Recurrent Unit model from the Sequence to sequence tutorial. It can respond to something like four sentences. When it comes upon a question that it doesn't expect it defaults to a certain sentence. It can answer questions that you might ask in a rudimentary conversation. It has answers to prompts like 'hi', 'How are you?' and 'What do you do?'. If you tell it your name it will tell you that its name is 'Sarah'. It doesn't answer arbitrary questions. It cannot answer 'What is your favorite color?'. It can not tell you the time. The default reply sentence for unknown prompts is 'Hi, how are you today?'

Questions

This is the sample question list as it was answered by the model.

```
> hello
hi , how are you today ?
> what is your name?
hi , how are you today ?
> what time is it ?
I like to read a lot
> what do you do ?
i'm a student
> what is your favorite color ?
hi , how are you today ?
> do you like red ?
hi , how are you today ?
> do you like blue ?
hi , how are you today ?
> what is your favorite candy ?
hi , how are you today ?
> do you like ice cream ?
yes , i do
> good bye
hi , how are you today ?
```

Checklist:

- ☑ All the responses are in plain English. There is no gibberish.
- ☑ There is a variety of answers. Not all answers are the same.
- ☒ Some of the answers are re-used and do not follow logically from the questions. The 'favorite color' and 'favorite candy' questions are nearly ignored. For those questions the model answers with 'Hi, how are you today?'. This seems to be the model's default answer.

☑ The model does not use 'No' or 'I don't know'.

☒ The model does not have an answer for 'Good bye'.

3.10.4 Smart Speaker - Transformer Model with Persona Corpus

The transformer model is installed on the Raspberry Pi. This model is more dynamic than the GRU hand-coded model.

The transformer model takes about two minutes to boot on the Raspberry Pi. After that the time between responses is slow. The time between the first two or three responses is uncomfortably slow. After those first responses the time between answers gets to be more natural.

There is one special tone that the Raspberry Pi gives at the end of loading the model. This tone notifies the user that the model is loaded and ready to respond to questions.

3.10.5 Chatbot - Transformer Model with Movie Corpus

Using the Movie corpus we trained a transformer model to use as a chatbot. This transformer was not pre-trained with any large corpus, so this example did not use transfer learning.

This model uses the tensorflow library, not Pytorch, and a transformer model that is larger than the other Transformer based model that uses the Persona corpus.

The memory footprint of the model while it was running was above 1.5 Gigabyte. The model could be installed on a Raspberry Pi 4B board but it requires a python package called 'tensorflow-model-server' and this package would have to be built from source for the Raspberry Pi.

The model took about seven days to train with a CPU based processor. The goal for training was 50,000 lines from the movie corpus. After training the loss graph was consulted and the installed version was culled from the saved checkpoint at the 45,000 line point.

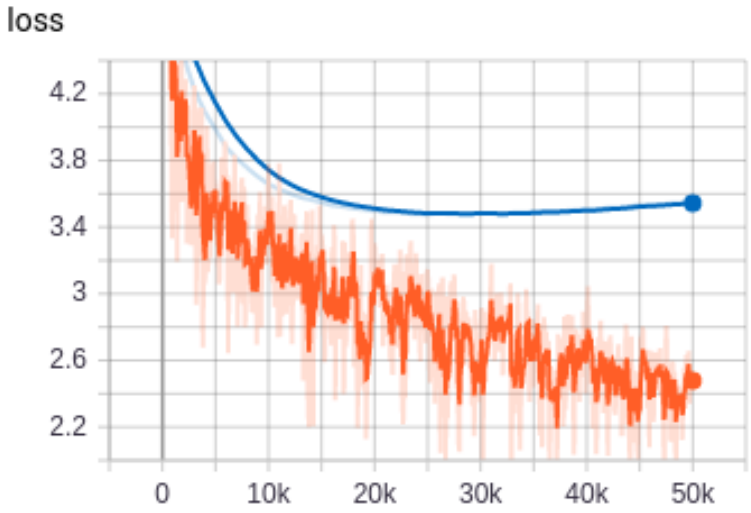

Figure 3.1: Loss - Orange is training loss and blue is evaluation loss.

Subjectively this transformer model is better than the Transformer model based on the smaller hyper-parameter set and the Persona Corpus.

Questions

This is the sample question list as it was answered by the model.

```
> Hello.
hello
> What is your name?
i don't know
> What time is it?
i don't know
> What do you do?
what do you mean ?
> What is your favorite color?
i don't know
> Do you like red?
no
> Do you like blue?
```

```
yeah
> What is your favorite candy?
i don't know
> Do you like ice cream?
yeah
> Good bye.
bye
```

Checklist:

- ☑ All the responses are in plain English. There is no gibberish.
- ☑ There is a variety of answers. Not all answers are the same.
- ☒ The 'favorite color' and 'favorite candy' questions are ignored.
- ☒ The model does in fact use 'No' or 'I don't know'. It also likes to answer 'yeah'.
- ☑ The model does have an answer for 'Good bye'.

3.10.6 Smart Speaker - Transformer Model with Movie Corpus

The transformer model is installed on the Raspberry Pi. It takes about five seconds to answer any question.

The transformer model takes about two minutes to boot on the Raspberry Pi. After that the time between responses is slow. The time between the first two or three responses is uncomfortably slow. After those first responses the time between answers gets to be more natural.

There is a special tone that the Raspberry Pi gives at the end of loading the model. This tone notifies the user that the model is loaded and ready to respond to questions. The model is also configured to beep intermittently during operation to signal that it is processing an input. This is helpful for a configuration where there is no monitor.

3.10.7 Chatbot - Generative Pre-training Transformer 2 Model

We used a pre-trained Generative Pre-training Transformer 2 model with the large english corpus to produce a chatbot and ascertain if this model works better than the sequence-to-sequence model. In our tests this worked well. The corpus is called 'WebText'.

For our experiments Generative Pre-training Transformer 2 was used for the chatbot model in 'zero-shot' mode. This means we did no special fine-tuning of the model in the application.

We did do some special coding for the input and output code in order to operate it as a chatbot. Output was limited to about 25 tokens.

Input to the model was prepended with the character string "Q:" by our code. Output was observed to have the character string "A:" prepended to it. We assume therefore that the model was at some point exposed to the "Question/Answer" paradigm in written passages during its training. This was helpful.

Output from the model was usually larger in size than we needed. Also, output had the character of having some sensible utterance followed by some output that was only a partial sentence.

It was necessary to 'scrape' the output. First the output was checked for the "A:" character string at the start. If it was there it was removed. Then the first complete sentence was used as output, while words and phrases after that were discarded.

Context Experiment

We decided that we would attempt to give the model some details that it could draw on during normal execution. We had two choices here. One choice was to train the model using fine-tuning and transfer learning to recognize certain questions and to supply answers. The other choice was to simply show the model the list of facts that we thought were important before every input sequence. This information would be summarized with each reply.

The second choice was more interesting. The text that the model was shown always included the name of the model (picked somewhat arbitrarily) along with information about the location of the model and the occupation. The time was also included.

This worked under most circumstances. Occasionally the model would report the wrong time when asked. This sort of activity is mysterious.

History Experiment

We tried concatenating all input and output and including it with each question. The hope was that the model would treat the 'Q' and 'A' as historical input and then it could refer to that input as it answered new questions. This didn't work, and the model was usually confused by the extra input.

Artificial Intelligence Markup Language Experiment

It was deemed helpful if the model could be given a question and instructed how to answer it. To this end AIML files were constructed and an AIML kernel was employed. The user's question was shown to the AIML kernel and then the model was shown the kernel's output (if there was one) along with the original question. The hope was that the output could be controlled by the AIML component. It didn't work. The AIML confused the model, and the model would not reliably choose to answer with the AIML text, as it might with the time of day.

Overall

Subjectively the model was the best of those tested. The model would answer questions about it's location, it's name, and the time, faithfully most of the time. Interestingly there where times when it did not do so. Some times it used alternative answers. For example, it would answer with the time but not the correct time. This was odd.

Under almost all circumstances the output was sensible English. There were no times where the model replied with gibberish.

The subject matter of the prompts did not need to be the same as the simple introductory conversation of the transformer model. In fact any subject matter could be chosen and the model would answer. The model did not remember its own answers but it was consistent. Questions it answered include 'What is your favorite color?' and 'Do you like lollipops?'.

Questions

This is the sample question list as it was answered by the model. Note that the information mentioned in the answer about the time was accurate when the test was run.

```
> hello
Hello.
> what is your name ?
My name is Jane.
> what time is it ?
02:59 PM January 28, 2020.
> what do you do ?
```

```
I am a student.
> what is your favorite color ?
I love the color of the rainbow.
> do you like red ?
Yes.
> do you like blue ?
I do.
> what is your favorite candy ?
I love candy.
> do you like ice cream ?
I do.
> good bye
Good bye.
```

Checklist:

- ☑ All the responses are in plain English. There is no gibberish.
- ☑ There is a variety of answers. Not all answers are the same.
- ☐ It is still debatable weather or not the answers to the questions about 'favorite color' and 'favorite candy' are good. The model could have a set of answers that it can use for this kind of question. The model seems to know what candy is and to a lesser extent what a color is. Some of the time the answer includes a word from the question sentence that would lead you to believe that this model has fewer stock answers. The answers are good but not perfect.
- ☑ The model does not use 'I don't know' that often.
- ☑ The model does have an answer for 'Good bye'.

The model will answer with it's name and you can tell it your name, but it is confused by this. It will on occasion tell you that it's name and your name are the same thing. This is in part because it cannot remember what it most recently said to you or what you most recently said to it.

3.10.8 Smart Speaker - Generative Pre-training Transformer 2 Model

Tests showed that the Generative Pre-training Transformer 2 chatbot worked well. We wanted to continue and allow the chatbot to have more of the abilities of a smart speaker. We constructed a simple corpus that contained key phrases that we wanted the chatbot to recognize and act upon. We did some transfer learning with this new corpus.

We found that one of two things would happen. The chatbot would either learn the new phrases and forget all it's pre-training, or it would not learn the new phrases and it would retain all it's pre-training. For our examples there seemed to be no middle ground. Comparisons were made with all available models and a version without the transfer learning was settled on.

Code was added that uses Text To Speech and Speech To Text libraries. In this way the model could interact with a subject using auditory cues and commands.

We did some programming that allowed the model to launch programs when directed to by the user. In this way we have tried to move our project closer to the smart-speakers that are produced commercially. The programming did not rely on the neural-network aspects of the model. Instead the code used heuristics and simple word recognition. This code can be disabled when the model is run from the command line.

The Raspberry Pi model that the Generative Pre-training Transformer 2 was installed on was the 4B with 4GB of RAM. It is largely for this model that we cross compiled the Pytorch Python library for the ARMv7. The GPT2 model fit on the Raspberry Pi. While execution on the production laptop was instantaneous, execution on the Raspberry Pi took about 13 seconds for every response from the neural network.

The Raspberry Pi was outfitted with a microphone and a speaker but no mouse, monitor, or keyboard. The program was modified so that there was a tone every time the model was ready for input. Without such a tone it would be difficult to know when to speak and when to wait for a response. Aesthetically this arrangement is not perfect, but it allows the Generative Pre-training Transformer 2 model to be physically installed on the Raspberry Pi.

3.11 Observation

It is important here to compare the GRU chatbot with the larger Transformer based chatbot. Using our subjective qualifications we see that the GRU model answers with more variety than the

transformer model. The important observation is that the hyper-parameter set for the Transformer model can be expanded and enlarged as needed before training. The GRU model cannot be trained successfully with an arbitrarily large hyper-parameter set. We can train a larger Transformer and obtain the benefit associated with this, namely better responses.

A single further observation is that the GRU model responds very quickly, while a transformer model may take more time relatively. This is not a problem for general applications, but for our purposes we cannot ignore the time spent by the transformer model when it is installed on a small computer like a Raspberry Pi.

The respective value of each of the models changes slightly when you consider what platform they will be implemented on. The GRU responds more quickly and so it retains more worth.

3.12 Tests

Turing Test

The Turing Test concerns itself with the question of weather a computer is intelligent. Turing says that intelligence is too hard to describe, and that if the computer can convince you that it is intelligent then it is.

Weather this is right is beyond the scope of this paper. The people who trained the Generative Pre-training Transformer 2 were apprehensive about their model's ability to generate human speech. At first when they finished their model they decided not to release the largest version to the public for several months (Radford et al.)[4]. Ultimately they did release their large model.

The creators of the model used it differently than our chatbot implementation. They generated paragraphs of text, and it was determined at first that the ability of the model to impersonate a human was too great. It was felt that the model could be used to spam facebook and other social networking sites with content that was very convincing. If the model could be used to convince people to act badly, then it should not be released. Humans are susceptible to the sentiments of those they see as their peers. If the model was, for better or worse, passing the Turing test, then it should not fall into the wrong hands. This was the concern of the coders at the time.

Ultimately the large model was released, either because the developers decided the model was not as good as originally estimated, or because they didn't care.

Winograd Schema

Winograd schemas are named after Terry Winograd. The idea is that there is a sentence presented that has two meanings. A computer finds these sentences challenging to understand, and that makes them interesting for the development of Artificial Intelligence.

An example follows.

He didn't put the trophy in the suitcase because it was too [big/small]

We can choose which bracketed term to use, and we must choose only one bracketed term. If we choose 'big' then we are referring to the trophy. If we choose 'small' then we are referring to the suitcase. Human beings can easily see the pronoun 'it' refers to either the suitcase or the trophy. Computers have trouble with these determinations.

The Transformer, and the Scaled Dot-product Attention that it uses, lends itself to discussion of Winograd schema. In the chat bot example, we are less interested in the Winograd example because it doesn't come up often. However, in the case of the Generative Pre-training Transformer 2, and it's exhaustive training, it is interesting to consider the Winograd style example sentences.

There is a Winograd Schema Challenge and something of a formula for constructing your own Winograd schema (Wikapedia contributors, 2020). [17]

Appendix A

Abbreviations

AIML Artificial Intelligence Markup Language

BERT Bidirectional Encoder Representations from Transformers

BPE Binary Pair Encoding

GPT2 Generative Pretraining Transformer 2

GPU Graphical Processing Unit

GRU Gated Recurrant Unit

NLP Natural Language Processing

RNN Recurrent Neural Network

SR Speech Recognition

Bibliography

[1] O. Vinyals and Q. V. Le, "A neural conversational model," *CoRR*, vol. abs/1506.05869, 2015.

[2] Matthew Inkawhich, "pytorch-chatbot," 2018. https://github.com/pytorch/tutorials /blob/-master/beginner_source/chatbot_tutorial.py.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[5] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[6] Denny Britz, "Recurrent neural network tutorial, part 4 – implementing a gru/lstm rnn with python and theano," 2015. http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/.

[7] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014.

[8] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[9] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*

(EMNLP), (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

[10] P. Mazaré, S. Humeau, M. Raison, and A. Bordes, "Training millions of personalized dialogue agents," *CoRR*, vol. abs/1809.01984, 2018.

[11] M. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *CoRR*, vol. abs/1508.04025, 2015.

[12] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[13] NVIDIA Applied Deep Learning Research, "Megatronlm: Training billion+ parameter language models using gpu model parallelism," 2019. https://nv-adlr.github.io/MegatronLM.

[14] Nemanja Milosevic, "Compling arm stuff without an arm board / build pytorch for the raspberry pi," 2019. https://nmilosev.svbtle.com/compling-arm-stuff-without-an-arm-board-build-pytorch-for-the-raspberry-pi.

[15] Erik Maciejewski, "Tensorflow serving arm - a project for cross-compiling tensorflow serving targeting popular arm cores," 2020. https://github.com/emacski/tensorflow-serving-arm.

[16] C. Danescu-Niculescu-Mizil and L. 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.

[17] Wikipedia contributors, "Winograd schema challenge — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Winograd_Schema_Challenge&oldid=938834078`, 2020. [Online; accessed 15-February-2020].