Capstone Project 2 Milestone Report

The Problem

The problem/challenge that this project aims to 'solve' is that of closed-domain *Question Answering (QA)*, a Natural Language Processing discipline concerned with building systems that answer questions posed by humans in natural language. Closed-domain QA refers to systems that are restricted to questions within some 'domain', as opposed to open-domain QA systems of which any question can be asked. QA systems are a popular research topic, and datasets such as the '*Stanford Question Answering Dataset'* (SQuAD) and *'bAbI'* from Facebook Research are popular dataset choices for closed-domain QA systems, and indeed the reason for their existence is to further develop QA systems (more on the dataset of choice for this project later).

Thus the task itself is to build a QA system. Furthermore, the project will serve as a personal challenge to develop in areas of Deep Learning and NLP.

Data Acquisition/Wrangling

The dataset of choice for the project is the '*Stanford Question Answering Dataset'* (SQuAD), a reading comprehension dataset, consisting of questions posed by crowd workers on a set of Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage. I chose SQuAD since, unlike bAbI, it is natural (non-synthetic) dataset. SQuAD consists of 150,000 question-answer pairs, 100,000 of those questions are 'answerable' and 50,000 are 'unanswerable'. The aim of the dataset is to encourage further research and development in the field of Question Answering and NLP. More can be found at: https://rajpurkar.github.io/SQuAD-explorer/. It's important to note that for each answerable question in the dataset, the answer is necessarily a continuous span of the context. This means that the question can be found in the context itself, and the continuous span means that the answer string is a single undivided substring of the context. This leads to strong implications that I will cover later.

The data is in a JSON file so the acquisition process is as simple as downloading the file and reading in the file via pandas. The goal of the wrangling step is to transform the data from the JSON format into a pandas dataframe where each row is a question, and the columns are: the question, answer, context of the question, if the question is answerable etc.

After analysing the JSON it was clear a simple import was not possible given the complex nested structure, hence the need for nested for loops. We looped over the Wikipedia articles present in the dataset, and then over the 'contexts' (paragraphs in the Wikipedia article), extracting the questions/answers (along with other fields) for each context into a dataframe and appending the dataframes to a 'master dataframe'. We also add the 'context' and 'title' of the Wikipedia articles as columns. The 'answer_data' function was defined in response to the differences in JSON format for the answerable and unanswerable questions to extract the 'start_index' and 'text' fields for the answer. As mentioned above, simply put, the data wrangling steps taken above can be best summarised by going over each theme and each context in turn and extracting the relevant information to create rows of data in a dataframe, the details of the implementation are a result of the structure of the JSON file.

Finally, to make the project more manageable we will only train and test our model on 'answerable questions', that is, we filter out 'context-question-answer' triplet rows for impossible questions.

Word Vectorization

I make an assumption at this point that the reader knows about word vectorization to some level. 'One hot encoding' as a word vectorization method, while simple, is not very 'useful' for many NLP tasks, as it does not encode any information about the relationships of words with one another etc. In the context of Question Answering this is especially important as the system relies on some kind of 'comprehension' of the questions asked of it to be able to provide an answer. To that end, after doing a moderate amount of research 2 main word vectorization 'methods' stood out, word2vec and GloVe. As also discussed in the notebook, given the slight edge in performance in most NLP tasks, we chose GloVe's pre-trained word vectors over word2vec's skip-gram word vectors.

GloVe has pre-trained vectors for various corpora including *Wikipedia 2014*, *'Common Crawl'* (a crawl/scrape of the World Wide Web) and *Twitter*. Given the differences in the style of language, amongst other aspects, it makes sense to use word vectorizations pre-trained on a corpus that is most alike the dataset we are using for this project (SQuAD). Given that the dataset is a bunch of Wikipedia articles along with questions and answers, it makes most sense to use GloVe's *'Wikipedia 2014 + Gigaword 5'* pre-trained word vectors. We will not re-train the word vectors with either method, as the file/corpus size (40MB) is not large enough to guarantee an increase in performance over the pre-trained variants.

In conclusion, GloVe is the initial word vectorization method of choice, we will use pre-trained word vectors, and in time, we will try out various word vector dimensionalities from GloVe and word2vec providing it is time efficient to do so (update: given the time constraints, we did not end up testing word2vec).

Choosing the Architecture / Building the Model

Firstly, we had to choose the model architecture and, as with the choice in word vectorizations, the choice to go with the *Dynamic Memory Network* neural network architecture was based on its state of the art performance on QA tasks (at the time of writing). *End to End Memory Networks* are also a good choice.

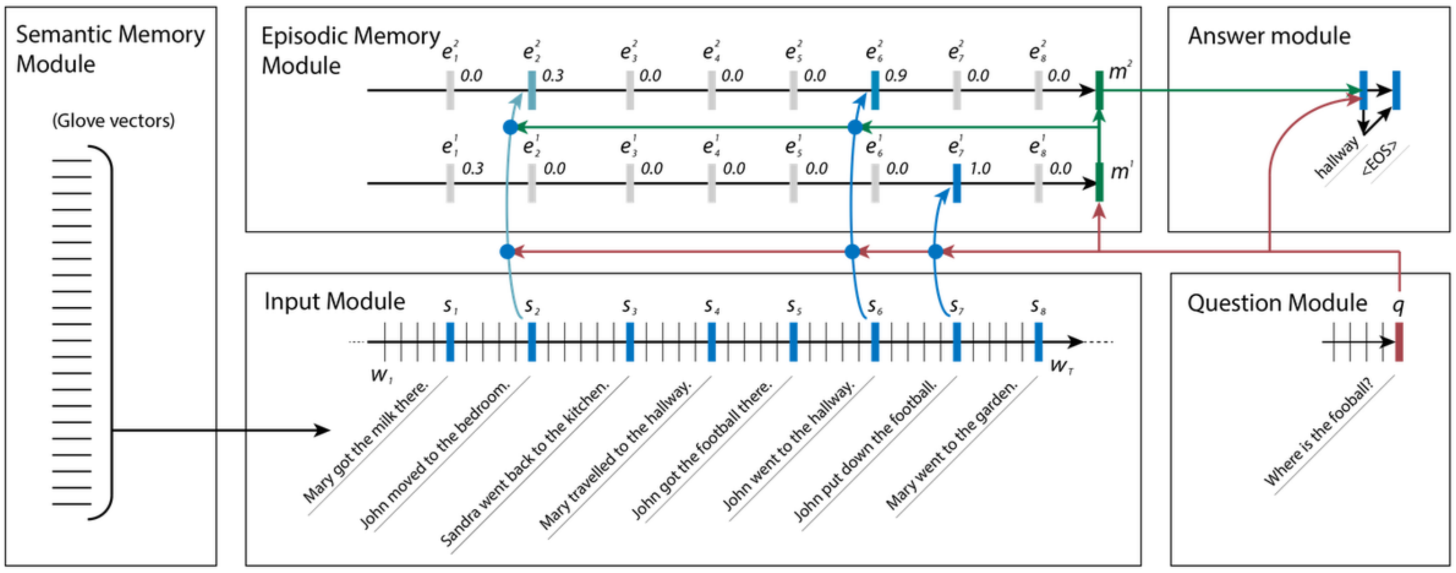

Illustration of DMN performing transitive inference.

For the purpose of being ‘on the same page’ with the reader I will briefly cover the basics of the Dynamic Memory Network (DMN) architecture. For a detailed description I highly recommend reading the research papers that introduced the DMN, which can be found at https://arxiv.org/pdf/1506.07285.pdf and https://arxiv.org/pdf/1603.01417.pdf as well as checking out the following YouTube videos of the creators discussing the DMN: https://www.youtube.com/watch?v=T3octNTE7Is&t=1s and https://www.youtube.com/watch?v=Qf0BqEk5n3o&t=1s .

Above is a visual representation of the DMN and its ‘modules’. The DMN is comprised of 4 modules: Input Module, Question Module, Episodic Memory Module and Answer Module. I did not include the Semantic Memory Module as its effectively the data input to the model. Describing the DMN in such a way allows us to ‘swap out’ parts of the architecture easily to allow the model to be used for various tasks. The example provided in the research papers is that of Textual and Visual Question Answering, although it’s important to note that the main idea conveyed by the creators of the DMN is that if we can find a way to describe our problem in a Question-Answer framework, then we can use the DMN to solve the problem (particularly so in the field of NLP). For example, Machine Translation can be thought of as a Question Answering problem (Q: “What is ‘Hello my name is Lukas’ in French?” A: “Bonjour mon nom est Lukas”). There is a lot more to be said about this idea, but the general takeaway is that the capabilities of the DMN are far greater than simply a specialised system used for QA.

The reason for going with keras (specifically keras’ functional API, using the tensorflow backend) is its moderate ease of use (as opposed to pure tensorflow), though the project did require the use of tensorflow for tensor manipulation operations. Let’s now discuss the meanings of the modules and how we implemented them in our project.

Semantic Memory Module

This module provides the input to the DMN in the form of word vectorizations (which by definition embed the word itself, as well as its meaning/relation to other words). This is important as without a meaningful representation of the input, the network wouldn’t function as required.

The *‘tokenize()’* and *‘get_embedding()’* functions are effectively the ‘Semantic Memory Module’ in our project, given a context, question or answer string, we get a vectorised/embedded representation of the input (I will use ‘vectorised’ and ‘embedded’ interchangeably as they refer to the same thing). Some pre-processing was done following the construction of the *‘train_df’* dataframe in which each row is a unique sample (unique ‘context-question-answer’ triplet) and the columns are: context (the Wikipedia paragraph string), question (the question string), answer (the answer string), answer_start (the character index of the context string where the answer begins) and is_impossible (Boolean value denoting if the question of the sample is possible to answer or not).

‘*Answer_start’* and *‘answer_end’* numpy arrays denote the start and end character indices of the answer string in the context. As mentioned above, this is an aspect of the SQuAD dataset and the major implication I alluded to above is that given these indices, the objective of our model (finding the answer to a question) can be modelled as training the model to predict the start and end indices of the answer within the context. This is referred to as ‘strong supervision’, and we are building a strongly supervised DMN.

Input Module

The purpose of the Input Module is to provide an embedded representation of the context (for all contexts). This means that the meaning and relationship of the words and sentences are captured by fixed length arrays. The module does so by taking in a context vectorization (matrix of numerical values) with shape (number of words in context, word vector size) and returning ‘facts’. A ‘fact’ is defined as an embedded representation of a sentence in a context (fixed length vector). So if our context has 4 sentences, the Input Module will output 4 facts.

The Input Module is implemented using a Bi-directional GRU (Gated Recurrent Unit). The recurrent nature of the GRU allows it to produce outputs that capture the meaning of a sequence of words (which is what we require when we input the sequences of words/tokens representing the context).

However, the GRU requires a 3D tensor input of shape: (batch size, number of timesteps/max number of words in context, word vectorization size). Thus, each sample must be of shape (max num of words in context, word vectorization size), since contexts are of varying length, we ‘pad’ each sample embedding matrix with 0.0s to achieve the required shape. Padding is defined as the process of extending samples with some value (usually 0.0 or 0) to achieve a specific input shape. However, having padded our input, we don’t want the GRU to process these 0s like normal input and this is the function of the Masking Layer. The Masking layer creates a ‘mask’ for each input, a mask is a matrix of the same shape as the input, with 1s in locations where there is a real input, and 0 where there is a padded value. This mask is passed to the GRU layer and as a result the GRU does not process the timesteps whose mask is made completely of 0s, that is, timesteps which were padded.

The GRU now iterates over each timestep (word in each context) and outputs the hidden state at each timestep. Technically, the fact is defined as the hidden state output by the GRU at the timestep corresponding to the ‘end of sentence token’, “.”. This is the hidden state we want to keep, and it represents the sentence embedding we talked about above. The ‘*get_facts()*’ function provides this ‘fact extraction’ functionality by filtering out timesteps corresponding to “.” tokens. The function is provided with ‘sent_end_idx_input’ which is a tensor containing the sentence end indices. The Reshape layer ensure that the output is a 3D tensor of shape (batch size, max sentence length/max number of facts, number of hidden units).

Question Module

Similarly, to the Input Module, the Question Module takes in questions (again as word vectorization matrices) and returns a question embedding representation. Given that each Question is one sentence long there is only one embedded representation. Once again, a GRU is used to produce the output, but this time a vanilla GRU suffices as we are only dealing with one sentence. Padding and masking of the Question Module input is performed in the same fashion as described in the Input Module section.

Episodic Memory Module

This is where things get tricky from all perspectives. In the interest of keeping the amount of text to a minimum I will skip over some details. The high-level idea of the Episodic Memory Module is to iterate over the facts returned by the Input Module (in light of the Question, represented by the output of the Question Module) multiple times, in order to output a memory which, we will pass to the answer module to produce an answer.

The module can be thought of as comprising an Attention Mechanism and a Memory update Mechanism. Attention is a huge topic and perhaps the magic of the dynamic memory network. It allows the network to exhibit ‘transitive reasoning’-like capabilities. It does so by having multiple iterations over the facts (multiple attention scores, multiple memories generated). There are various implementations and definitions of attention mechanisms (soft attention, GRU based attention). I implemented most of the attention module, but considering the custom hidden state output calculations required by the Attention GRU, I realised that I would have to write my own keras layer (which comes with its own challenges and learning curves) to implement the Episodic Memory Module. Given the timeframe in which this project had to be completed, I decided to use an implementation already available from another GitHub user. This provided me with an effective Episodic Memory Module layer that I could use in my computation graph. Under the hood it's an AttentionGRU layer (a vanilla GRU layer with an 'attention based' hidden state update calculation) used in an iterative manner to compute memory vectors, the last of which is the output of the Episodic Memory Module. Full code can be found in the repository of this project.

Answer Module

The Answer Module generates an answer given the output vector from the Episodic Memory Module. Once again there are many kinds of answer modules. For single word answers, a single layer feed forward neural net with softmax activation is a viable option, however given that the SQuAD dataset contains answers that have varying length, we cannot use this approach. The approach suggested by the research paper is to use a GRU for multi-word predictions.

We ended up going with 2 separate single layer feed forward neural nets with softmax activation. They output for each neural net is a matrix of shape (batch_size, max character context length). Given the softmax, each position in the matrix can be interpreted as the probability that the start or end index of an answer of a given sample is on that character index in the context.

Training and Testing

We use categorical cross-entropy as our loss function. To do this, we 'one hot encode' our *answer_start* and *answer_end* indices to vectors of length (max character context length) where all the values of the array are 0 other than one index which has a value of 1 corresponding to the index of the start of end of the answer. In doing so the model aims to minimise the difference in predicted probability of the start/end index of the answer and the probability of the true class.

We then choose the start and end indices with the highest probabilities and slice the context of that sample with those indices to obtain an answer.