**Introduction**

The Department for Education (DfE) is committed to supporting the [AI Opportunities Action Plan](https://www.gov.uk/government/publications/ai-opportunities-action-plan/ai-opportunities-action-plan). Most recently, the department has become strongly focussed on the delivery of Generative artificial intelligence models to facilitate teaching and learning, helping to alleviate the learning facilitation burden experienced by many teaching professionals.

The objective of this project is to develop a prototype which demonstrates how such technology could potentially be leveraged in a classroom setting. One of the most pervasive issues, and a major cause for concern for the DfE is the continuing struggle to support STEM subjects beyond the standard curriculum. However, the ability of AI to make a tangible difference in this area is fast becoming apparent, and it is the application of such tools, together with a comprehensive evaluation, to which this project will focus.

More specifically, we will concern ourselves with the development of a prototype **maths question and answer teaching support assistant**, utilising **Sequence-to-Sequence** (**encoder-decoder**) modelling alongside ***Long-Short-Term Memory*** (**LSTM**) models, trained on a variety of mathematics question and answer samples. The approach will include a defined data preprocessing pipeline, a robust model training regime employing ***k-fold cross-validation*** to avoid overfitting, a comprehensive ***hyperparameter tuning*** regime to maximise performance, and robust evaluation utilising ***Edit distance*** and ***BLEU scores***.

The intention here is not to create a first-class, production ready system, but rather to demonstrate and evaluate the potential benefits such an AI model and approach can bring to this learning need. Based on this study, conclusions will be drawn and recommendations made on those areas that offer potential for further development.

**Data Description**

Include EDA findings

Look at the sort of corpus the DfE would be interested in in terms of teaching kids

Why is this type of learning important, i.e. worded math questions

Exploratory Data Analysis (EDA) for our encoder-decoder models focuses on understanding the characteristics of the question-input, and answer-output pairs before training. Several techniques were applied both before, and during the training and model iteration phases.

1. \*\*Sequence Length Analysis\*\* – Examine the distribution of input and output sequence lengths to optimize padding and truncation.

2. \*\*Token Frequency Analysis\*\* – Identify common and rare tokens in the dataset to improve vocabulary selection and handling of out-of-vocabulary tokens.

3. \*\*Attention Visualization\*\* – If using attention mechanisms, analyse how the model focuses on different parts of the input sequence.

6. \*\*Alignment Patterns\*\* – Examine how input sequences map to output sequences, especially for tasks like translation or summarization.

7. \*\*Noise and Missing Data Handling\*\* – Detect inconsistencies, incomplete sequences, or irregular patterns that may impact training.

**Methodology**

Show evidence as to why we made this choice of model, i.e. links to articles etc justifying why this would be a good choice…………..

There are a number of techniques we could employ to create our prototype, not least the …..blah blah

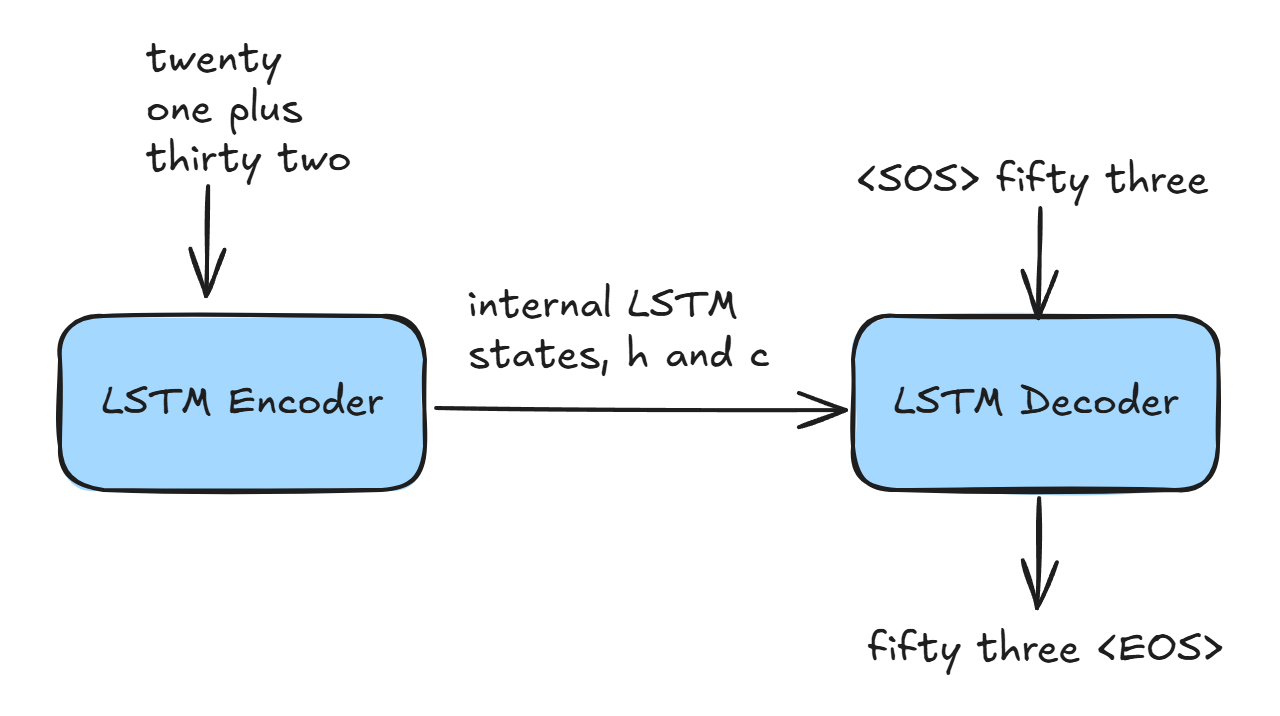
**Choice of Model**

The approach taken leverages a ***Neural machine translation*** (**NMT**) approach that employs deep neural networks to perform machine translation from the source question to the target answer. This requires the neural translation mechanism to take in the question as text (that is, the source language) as a sequence of inputs and encodes these to a hidden representation, which is then decoded back to produce the translated text sequence as the target answer. One of the key advantages of this NMT system is that the whole machine translation system can be trained from end-to-end together.

***Recurrent Neural Networks*** (**RNNs**) architectures such as ***Long Short-Term Memory*** (**LSTMs**) and/or ***Gated Recurrent Units*** (**GRUs**) are the favoured models used in neural translation machine architecture. We will leverage the power of LSTM models as the basis of our study given their ability to handle sequential data by remembering important information while selectively forgetting irrelevant details. Unlike traditional models that struggle with long-term dependencies, LSTMs generally excel at processing time-series data, such as predicting words or patterns in text. By maintaining context over time, LSTMs enable smarter predictions and more accurate understanding of sequences, thus forming the basis for our prototype.

**Model Overview**

Our prescribed approach adopts an ***encoder-decoder*** model (or what is often referred to as ‘**sequence to sequence**’), with the training described using the following diagram which shows the high-level architecture of our proposed neural translation mechanism,



From this, we can see that the encoder LSTM is responsible for encoding the input question sequence into hidden states and memory cell-states . These hidden states and cell states [,] are intended to capture the context of the whole input sequence, therefore [,] represents a reasonable candidate on which the decoder network can be conditioned. This hidden and cell state information, [,], is passed to the decoder LSTM model as the initial hidden and cell states. The decoder is then trained on the target answer sequence, with the input target question sequence being one step behind the output target answer sequence. As per the decoder, the first word of the input question sequence is the token word <SOS> which represents ‘start-of-sequence’, while the output label is the word *fifty*. The decoder network is just trained as a generative language model, where at any time step , the output label, is just the next word with respect to the input, that is, . The only new thing is that the final hidden and cell states of the encoder (that is, [,]) is fed to the initial hidden and cell states of the decoder to provide content for the translation.

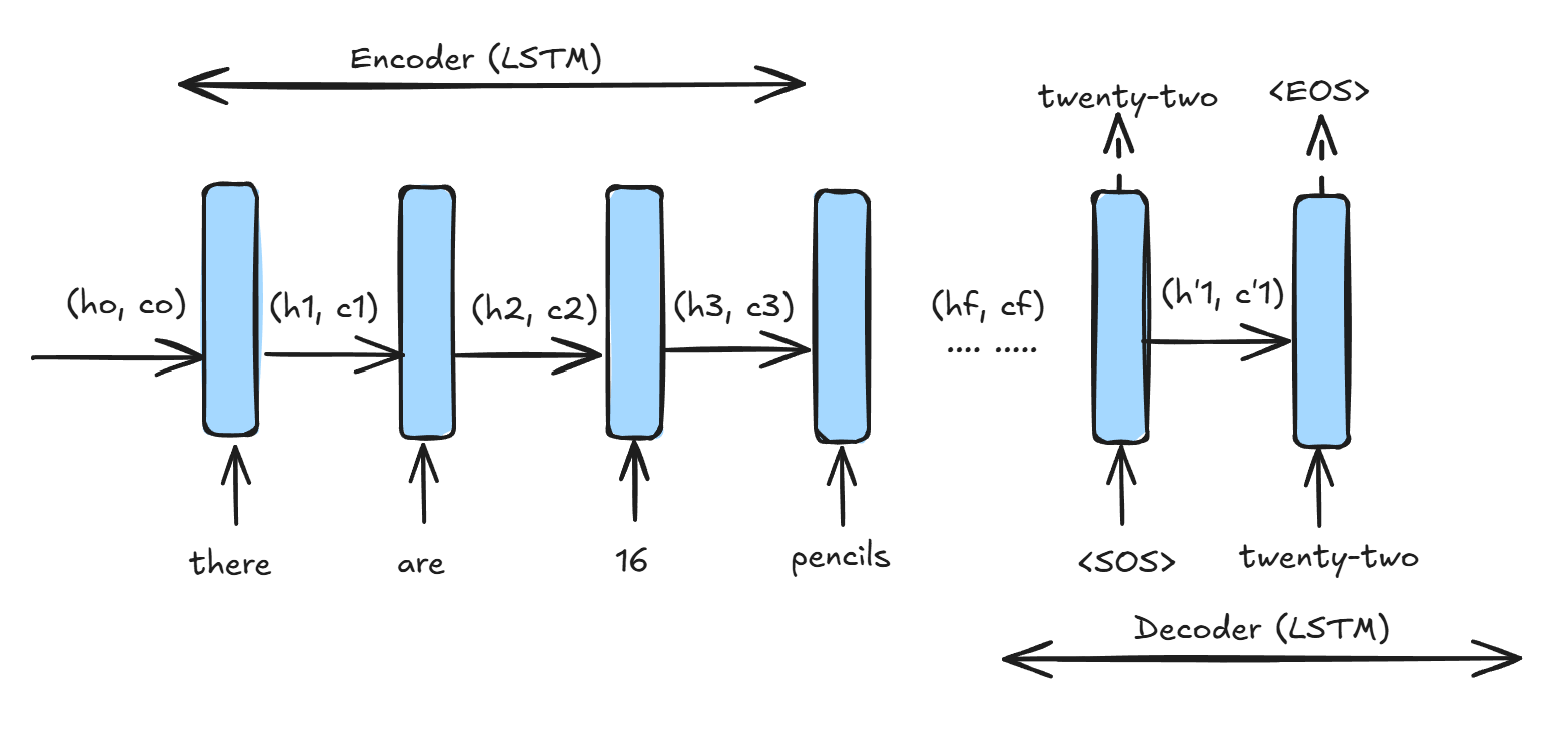
This means the training process for our math assistant can be thought of as building a language model for the target language, in our case the math answers represented by the decoder, conditioned on the hidden states of the encoder that represent the source language, in our case the math questions.

**Model Training Overview**

In the simplest terms, what we are trying to achieve when training the model is to make it as accurate as possible when predicting the next word in the target language translation, given the source text and the previous words it has already translated. This involves adjusting the model's internal parameters, , so that the probability it assigns to the correct next word is as high as possible at each step. Essentially, it's like teaching the model to step through the translation process one word at a time, constantly improving its understanding of the relationship between the source and target languages.

To summarise, the encoder will process the source input question sequence through an LSTM and encode the source question text into a meaningful summary. The meaningful summary is stored in the final sequence steps hidden and cell-states and . These vectors together ([,]) provide a meaningful context about the source input, and the decoder is trained to produce its own target sequence conditioned on the hidden and cell state vectors [,].

The following diagram gives a more detailed view of the training process associated with our math word problem assistant model. The input sequence *“****there are 16 pencils… ...****”* is converted to a meaningful summary through the LSTM, which is then stored in the hidden and cell-state vectors [,]. The decoder is then made to generate its own target sequence (**Answer**), conditioned on the input source from the encoder, through the information embedded [,]. The decoder at time step is made to predict the next target answer, that is, the word at time step , given the source input (**Question**).

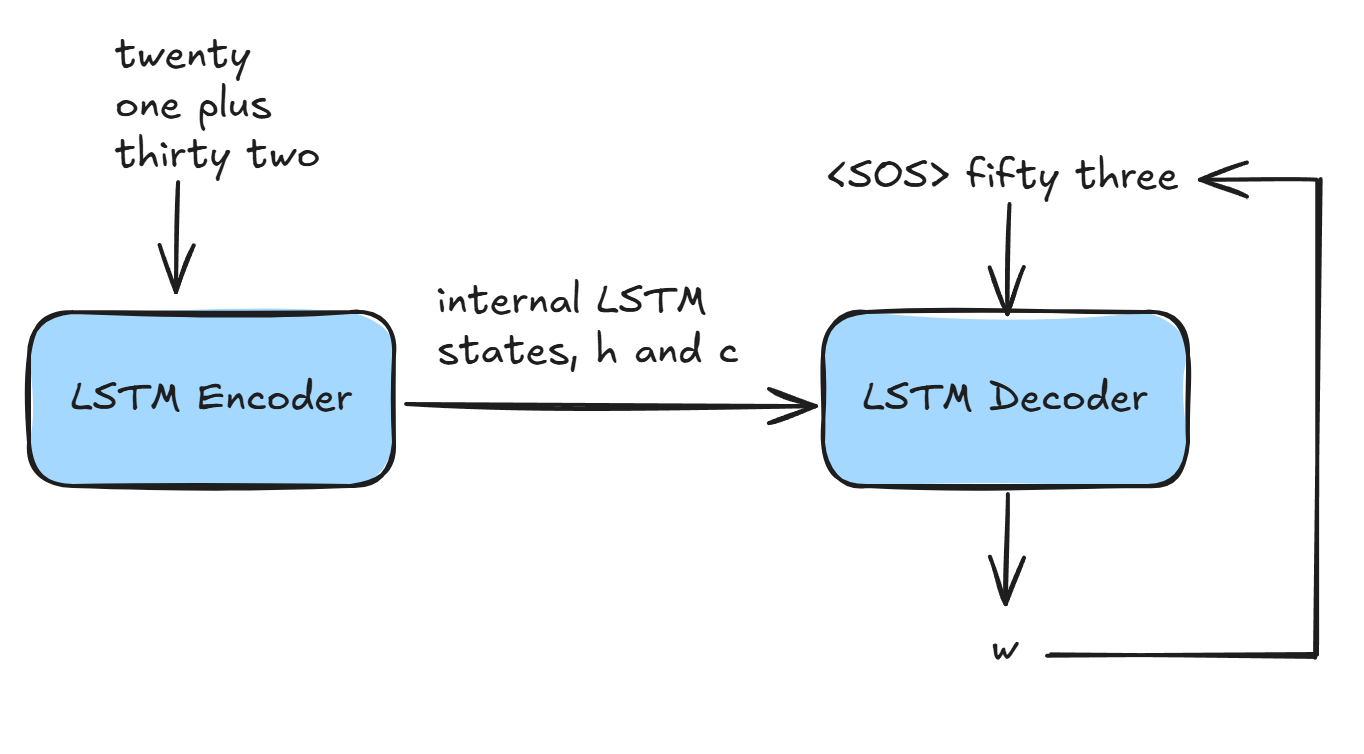


This is why there is a one-step lag between input sequence (**Question**) and target sequence (**Answer**). For the first time step, the decoder doesn't have any prior words in the target input sequence, and so the only information available to predict the target sequence (**Answer**) is the information encoded in [,]that is fed as the initial hidden and cell-state vectors. Like the encoder, the decoder also uses an LSTM and as discussed, the output target sequence is ahead of the input target sequence by one time-step.

With training complete, how can we use the trained model during infer an answer to our math-based question?

**Model Inference Overview**

The architectural flow for running inference on the model is a little different than that previously considered for training. During inference, the input sequence is fed to the encoder network and the final hidden and cell state produced, [,], is fed to the decoder’s hidden and cell-states. The decoder is converted into a single time step, and the first input fed to the decoder is the <SOS> token. So, based on [,] and the initial token <SOS>, the decoder will output a word, w, and also new hidden and cell states, [,]. This word is fed to the decoder again with the new hidden and cell states, [,], to generate the next word. This process is repeated until we encounter an end-of-sequence <EOS> token. The following diagram demonstrates this high-level architecture.



Our intention is to build a neural machine translation system that will learn to translate simple mathematical questions in sentence form and produce a correct answer, again in word form. We cannot directly feed our question and answer (Q&A) text data directly into our proposed model, since neural networks can only understand numerical values. As such, we will treat each word as an indexed value uniquely assigned to each given word in the vocabulary, the length of this index will be equal to the number of words present in each corpus. If the Q&A combination contains 120 unique words, we will have a vocabulary index of size 120, plus any additional standardised tokens we care to add. This tokenizer is required to read through all the question-and-answer data provided within the corpus and subsequently index each unique word to form a vocabulary the model can read. For example, let's assume that in the text corpus, we have the following question and answer, “***there are 16 pencils in one box and 6 pencils in another box - how many pencils are there altogether? twenty-two***” then we can define the indices for each of the words as follows,

|  |  |
| --- | --- |
| **Word** | **Index** |
| *<SOS>* | 0 |
| <*EOS*> | 1 |
| *Altogether?* | 2 |
| *many* | 3 |
| *pencils* | 4 |
| *box* | 5 |
| *there* | 6 |
| *are* | 7 |
| *and* | 8 |
| *another* | 9 |
| *twenty-two* | 10 |
| *there* | 11 |
| *-* | 12 |
| *how* | 13 |
| *one* | 14 |
| *in* | 15 |
| *16* | 16 |
| *6* | 17 |

So, if we consider the input question, we will have a sequence of words represented as a vector of indexed values. The next obvious question is how to manage the sequence length, since this might vary. The most accepted approach is to have a fixed sequence length either equal to the maximum sequence length of the sentence in the corpus, or a predetermined reasonable length. We will be using the answer word sequences twice, once as the output sequence of translation from the decoder, and once as the input to the decoder, with the only difference being that the output sequence will be ahead of the input sequence by one time step. So, the first word in the input target sequence would be the token <**SOS**> representing the ‘Start-Of-Sequence’, while the last word in the output target sequence would be the token <**EOS**>, marking the ‘End-Of-Sequence’.

If the target question sequence is as before, the input question and the output answer sequence in the decoder would be as follows,

Question: [**<SOS>**], [***there***],[***are***],[***16***], [***pencils***], [***in***], [***one***], [***box***], [***and***],[***6***], [***pencils***], [***in***], [***another***], [***box***], [***-***], [***how***], [***many***], [***pencils***],[***are***], [***there***], [***altogether?***]

Answer: [***twenty-two***], [**<EOS>**]

This in turn is translated into the required index vectors, which for our example would be as follows,

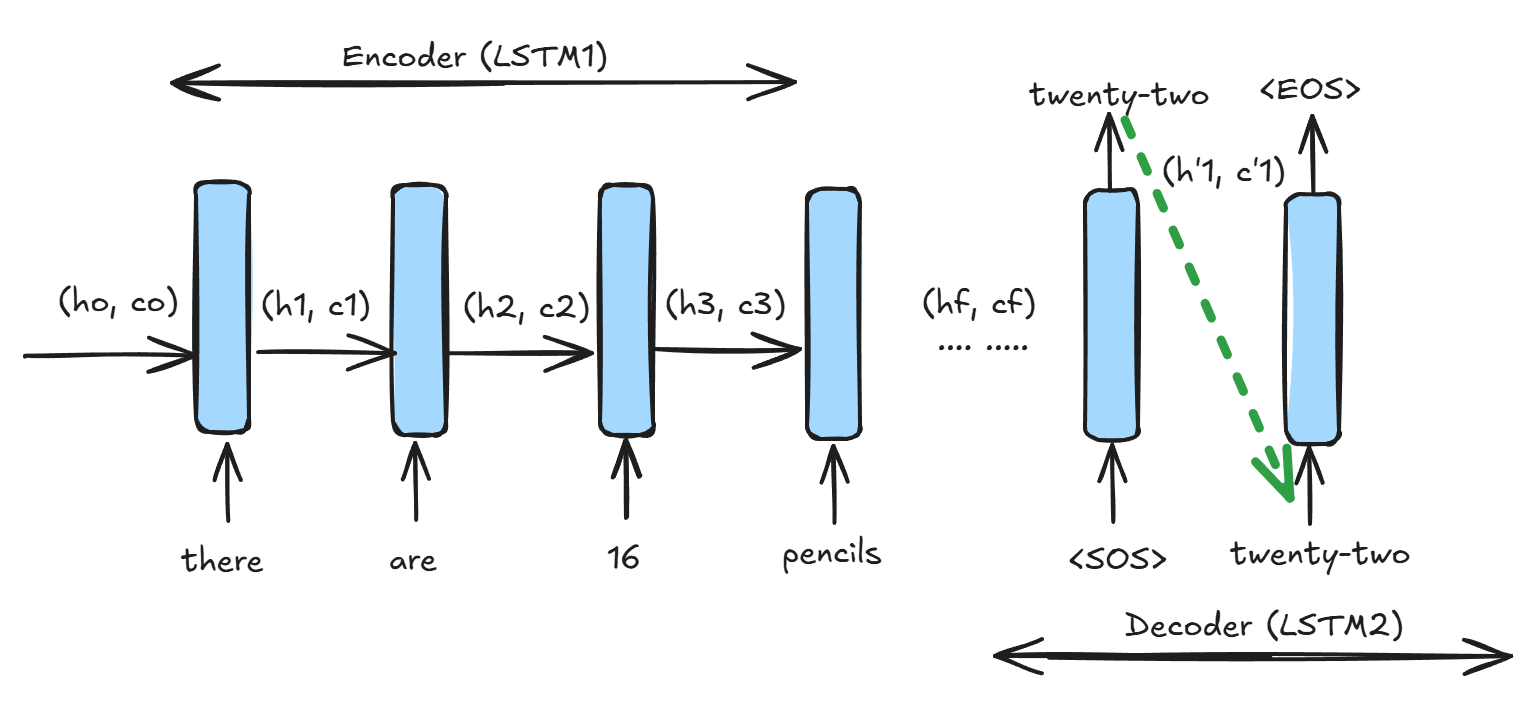
Question: [**6**,**7**,**16**,**4**,**15**,**14**,**5**,**8**,**17**,**4**,**15**,**9**,**5**,**12**,**13**,**3**,**4**,**7**,**11**,**2**]

Answer: [**10**]

Since our input question sequences can vary in length, we use a padding token <PAD> to ensure our sequence lengths are uniform, which helps to make batch processing more efficient. In this respect, we define a maximum sequence length and add the pad token to all sequences that fall under this prescribed standard length. These tokens are ignored during training using attention masks, preventing unnecessary computations. This technique ensures proper alignment and maintain the integrity of sequence relationships, as well as offering the potential to improve model performance by preventing bias toward shorter sequences while optimizing computational efficiency.

Our model can make answer predictions by leveraging the cross-entropy loss for each target question-and-answer in the model sequence. The actual question and predicted answer can be any of the words in the corpus that we have provided. The target label at time step would be an encoded vector, while the predicted output would be in the form of probability for each of thewords in the vocabulary defined by the corpus. We get the loss for the entire sequence by summing up the losses over all the sequence time, and since we work with mini-batch stochastic gradient descent, the average cost for the minibatch can be obtained by averaging the loss over all the sentences in the mini-batch. The mini-batch cost is used to compute the gradients for the stochastic gradient descent.

If we recall the working mechanisms of trained model, the encoder part of the model should work by taking question text sequences as input and providing the final hidden and cell-state vectors, [,], as output to the decode. Since we can't use the decoder network as is, since the target question input words are no longer fed to the decoder as was the case with training. Instead, we reduce the decoder network to consist of a single step and provide the predicted answer from that step as input to the next step. We start with the token <SOS> as the first input word to the decoder, along with [,], serving as its initial hidden and cell-states. The target output word *w1* and the hidden and cell state [,] generated by the decoder with [START] and [,] as the input is fed back to the decoder to generate the next answer, and the process repeats until the decoder outputs the dummy word <EOS>, in our case this step we immediately output the <EOS> token given our fixed answer length. The following diagram illustrates the stepwise representation of this inference procedure,



From above, we can see the output of the first step of the decoder is the predicted output “twenty-two”, while the hidden and cell states are [,]. This is fed to the decoder again, as shown by the green dotted line, to generate the next answer response, along with the next set of hidden and cell-states. The process is repeated, until the decoder outputs the dummy end character <EOS>, in our case we have a single step since we have fixed the answer output to a single word response (i.e. we generate a hyphenated response).

\*\* Mention the attention mechanism \*\*

\*\* Mention the dropout rate \*\*

**Data Preprocessing**

Data preprocessing for our sequence-to-sequence model involves several key steps to ensure the model effectively learns from structured question-answer pairs pairs. From our analysis so far, we have touched upon several model data requirements,

1. Reading the input files for the source (**Question**) and target (**Answer**) data corpus;
2. Building the vocabulary from the source and target texts;
3. Processing the input **Question** and **Answer** corpuses to their numeric representation so that they can be used in the neural machine translation network.

Of course, our first step is to read the generated question and answer data corpus from the file system. Since the data corpus has been generated as a comma separated file of question-answer pairs, with each column identifiable from their respective ***Question*** and ***Answer*** labels, it is a reasonable trivial task to load the required question and answer data into a workable format. In this respect, the math problem sequences are loaded from a named CSV file using the ***Pandas*** library. Question statements are extracted alongside their corresponding answer statements, and these are returned as separate labelled ***question\_sequences*** and ***answer\_sequences*** lists, making them accessible for tasks like tokenization and embedding conversion. When the question-and-answer data load function executes, additional validation mechanisms and data preprocessing techniques are utilised to further ensure the integrity of the dataset.

Error handling ensures that potential issues, such as missing files, absent columns, or unexpected errors, are gracefully handled. If the CSV file is not found, the function catches the expected ***FileNotFoundError*** and provides a clear message while returning empty lists early termination. We also look to verify the existence of the required ***Question*** and ***Answer*** columns before proceeding, raising a ***ValueError*** if corresponding fields are missing. Furthermore, missing values are handled by dropping rows containing ***NaN*** values in essential columns, ensuring the data remains clean and consistent. A general exception handling block captures unexpected errors, preventing any other unforeseen issues from disrupting execution.

Data cleaning is also applied to each question-and-answer text pair using a dedicated function that leverages a regular expression pattern designed to filter out unwanted characters from the question-answer text input while preserving essential elements. This expression allows lowercase text and numeric ranges ***a-z*** and ***0-9***. Additionally, the basic mathematical operators (***+, -, \*, /, =***) and question mark are included in the set, allowing expressions containing these symbols to be retained. The space character is also permitted to maintain readability and word separation.

The next step is to tokenize the question-answer data corpus and generate a vocabulary from this that our model can understand. This tokenization process involves converting our raw data question and answer text corpus into numerical tokens that our model can understand and process. In the context of our problem-space, tokenization involves splitting our question-answer corpus into distinct word groupings and mapping each word to a unique index value, essentially building a vocabulary which translates each known word into its corresponding index allowing each word to be uniquely identifiable. For this study, we have chosen to adopt this simplistic, although there are obviously more advanced approaches such as ‘***Byte Pair Encoding***’ (**BPE**) or ‘***WordPiece Encoding***’, which may prove more efficient and elegant but are beyond the scope of this report.

Practically speaking, our function tokenizes our question-answer data corpus while dynamically constructing the associated vocabulary. Initially, our dictionary only contains our specially defined tokens such as <SOS>, <EOS>, and <PAD> along with their respective indices. In particular, the <SOS> and <EOS> tokens are added to define input-output boundaries that allow the model to understand where a question-answer pair processing sequence starts and ends. As the function processes each comma-separate question-answer pair, it converts each unique word into its lowercase representation, essentially creating the basis for an individual token representation. If a word is not already present in the dictionary, it is assigned a new index based on the current vocabulary size, ensuring that each unique word is mapped to a distinct index value. This approach allows the model to continuously expand its vocabulary as new words are encountered. What we generate is a split between tokenised question and answer lists, recognisable as ***input\_data*** and ***target\_data*** labels respectively.

However, it is important to note that this approach of dynamic vocabulary expansion comes with potential challenges, such as inconsistent indexing across different training sessions, making it harder to ensure reproducibility which is certainly an issue that must be addressed if further development is undertaken beyond this analysis. As well as affording the ability to tokenize data for the model to understand, we must also provision a means by which to convert the model output (answer predictions) into a human readable form, and this is done by creating a reverse mapping dictionary to convert our generated token indices back into their associated text format. Once the sequences are tokenized, they are converted into ***PyTorch*** tensors using list comprehensions, making them ready for batch processing by our model. Since sequences can vary in length, dynamic padding is applied using the native ***PyTorch*** ***pad\_sequence()*** function to ensure uniformity. The padding operation extends shorter sequences to match the longest sequence in the batch, using a predefined <PAD> token. This step is crucial for maintaining consistency during training.

The preprocessing pipeline also defines a custom ***PyTorch*** ***Dataset*** class, ***MathWordProblemDataset***, which provides functionality for retrieving dataset size and accessing individual input-target pairs, making it useful for our structured batch processing approach. Finally, the dataset is integrated into a ***DataLoader***, allowing for seamless batch loading. Overall, this data preprocessing pipeline is designed to ensure that our math problem and answer dataset is properly tokenized, padded, and structured for our specific application. particularly in sequence-to-sequence models. Implementing additional enhancements, such as attention masking or tokenization refinement, could further optimize its effectiveness.

*\* Mention data preprocessing in terms of padding and adding the hyphen between answer \**

**Initial Model Training and Evaluation**

We need to mention the inclusion of dropout and attention used within our model

**Model Refinements, Tuning and Evaluation**

Xxxxxxxxxxxxxxxxxxxxx

**Results**

Discuss results and any conclusions and recommendations

**Conclusion**

Here is my conclusion

**References**

Learn to Add Numbers with an Encoder-Decoder LSTM Recurrent Neural Network

By Jason Brownlee on August 27, 2020 in Long Short-Term Memory Networks

https://machinelearningmastery.com/learn-add-numbers-seq2seq-recurrent-neural-networks/