|  |  |
| --- | --- |
| **Level 5 AI and Machine Learning Bootcamp – Final Project** | |
|  | |
| **Student:** | **Spencer David O’Hegarty** |
| **Assessor:** | **John McKechnie** |
| **Hand in Deadline:** | **11th April 2025** |

**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 comprehension 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**

The Department for Education (DfE) emphasizes the importance of mathematical word problems and comprehension because they are crucial for developing problem-solving skills, mathematical reasoning, and overall literacy in mathematics. Indeed, the DfE guidance on Mathematics GCSE states that GCSE mathematics examinations should enable pupils to “***comprehend, interpret and communicate mathematical information in a variety of forms appropriate to the information and context***”.

**Source:** https://www.gov.uk/government/publications/national-curriculum-in-england-framework-for-key-stages-1-to-4

Word problems require students to translate real-world scenarios into mathematical equations, fostering critical thinking and analytical skills. This sort of skill is expected to be built from **KS1** through to **GCSE**, with the DfE guidance on mathematics stating that teaching literature will be expected to include '***Language focus***' features to provide suggested sentence structures for pupils to use to capture, connect and apply important mathematical ideas. Likewise, at **GCSE** the guidance states that **GCSE** mathematics examinations should enable pupils to comprehend, interpret and communicate mathematical information in a variety of forms appropriate to the information and context.

Clear emphasis is being placed on this aspect of mathematical learning to boost a pupil’s ability to apply critical thinking both within the mathematical sphere, and beyond. As such, the development of such mathematical comprehension problems will form the basis for our proposed dataset, with focus on understanding the characteristics of the question-input, and answer-output pairs before training to ensure optimal performance as well as providing intuitive questions which align with the DfE’s proposed ideology in this context.

To keep things relatively simple, we will focus on producing a typical question-answer dataset posed to KS1 pupils. A typical **KS1 (Key Stage 1)** maths word problem is designed to help young learners apply basic arithmetic in real-world scenarios, much like the following,

*"****Sophie has 5 apples. She buys 3 more apples at the shop. How many apples does Sophie have now?****"*

This type of problem encourages children to use addition to find an answer to this problem. On this basis, a function has been developed to allow auto-generation of math comprehension problems based on several templated scenarios. The executed result produces a dataset of math problems and their corresponding solutions (addition and subtraction scenarios) which is stored in a CSV file format for further use. It does this by dynamically constructing word problems using predefined templates and filling them with random numbers. The logic ensures that problems will involve a random mix of addition or subtraction questions, depending on the phrasing of the question from the randomly chosen template. To make the answers more readable, the code uses the ***Inflect*** library to convert numerical answers into their word form and replaces spaces with hyphens for consistent answer formatting. For example, the problem type produced will be dependent upon the randomly selected template together with the appropriate mathematical operation based on keywords present in the question. If the problem involves verbal concepts like "***added***" "***altogether***" or "***total distance***" it applies addition. If the problem mentions "***loses***" or "***left***," it performs subtraction, with the computed result then transformed into its word equivalent.

Overall, this data generation pipeline is required to offer problems that vary dynamically and use appropriately structured language of reasonable length and form. To ensure data integrity, several techniques were applied both before, and during the training and model iteration phases.

1. ***Sequence Length Analysis*** – allowing us to examine the distribution of question-and-answer sequence lengths with a view to optimise padding;
2. ***Bigram (two-word phrase) frequency analysis*** – allowing us to identify common two-word phrase tokens in the dataset which will allow us to potentially focus on the models learning attention;
3. ***Attention Visualisation*** – since we have incorporated attention mechanisms (a model component that dynamically focuses on relevant parts of question data, allowing improvements in contextual understanding by assigning varying importance to different elements in the sequence), we are able to visualise how the model focuses on different parts of the question sequence during training.

**Sequence Length Analysis**

The prescribed process performs data analysis and visualization on each randomly generated dataset, with questions and answers analysed to determine their respective word count, the detail of which are further extrapolated to provide insight into the length of each sequence. This is achieved by splitting each text entry into words and computing the number of elements in those splits. Once the word counts are obtained, we can calculate basic statistics, including the average, maximum, and minimum lengths for both question-and-answer type. These metrics have helped greatly in the refinement of the data generation process, not least in ensuring that the question-and-answer datasets produced are reasonably structured and consistent, not least to reduce the need for artificial consistency measures such as data padding to ensure compatibility with model training requirements.

For example, if we consider the sequence length analysis results for a typical dataset generated by the final iteration of the data generator, clearly the questions lengths are reasonably consistent.

A graph with numbers and a bar

AI-generated content may be incorrect.

Of the five question types generated, their lengths vary between 16 and 21 words, which we consider acceptable for performance. There is a skew in the number of question frequencies generated for the last question so this may have the potential to cause training anomalies, however, we will retain this structure in order to test attention further. Although this picture looks reasonable, earlier iterations of the data generation logic applied a larger number of templates with a broader narrative which resulted in quite large skews between question types and the number of words generated; thus, encouraging the tokenisation process to apply an excessive number of padding tokens. The relative increase in the use of padding tokens can not only increase computational cost but can also affect model accuracy by forcing attention on meaningless tokens rather than actual content. By analysing the distribution of sequence lengths, we hope to ensure our model can minimise redundant padding by establishing reasonable maximum lengths based on the typical question-answer data corpus size. Additionally, batching sequences of similar lengths together should help to reduce inefficiencies, as our model will process data more effectively when padding is kept to a minimum. It is expected that an understanding of our question-answer sequence length distributions will lead to significant improvements by minimising the use of such filler tokens.

**Bigram (two-word phrase) frequency analysis**

The Bigram frequency analysis we have employed helps us to examine how often two-word phrases (bigrams) appear in our generated question datasets, which in turn helps us identify common word pairings, linguistic patterns, and contextual relationships between words which we can later use for attention analysis during training. If we consider the bigram frequency results generated by our training dataset, clearly, we have several two-word combinations that may attract strong attention during model training.

A screenshot of a graph

AI-generated content may be incorrect.

For example, frequent bigrams like "***how many***", "***in total***", “***he loses***” etc would feasibly be expected to draw attention during training, with the potential to facilitate correct answer predictions. This specific analysis will combine with the attention visualisations generated during training under the section headed, ‘***Initial Model Training and Evaluation***’.

**Attention Visualisation**

Since we have incorporated an attention mechanism in our sequence-to-sequence model (to be discussed in more detail in the section ‘**Model Training Overview**’), we can visualise this attention which should help us interpret how our model focuses on different parts of a given question sequence while generating the answer. If successful, we should be able to derive insight into our model’s decision-making process.

In this respect, we adopt the common approach of using attention heatmaps, where rows represent words in the input sequence and columns represent words in the output sequence. The intensity of the colour at each intersection indicates how much attention is given to a particular questions word when producing an answer output. As before, we will evaluate the attention under the section headed, ‘***Initial Model Training and Evaluation***’.

**Methodology**

There are of course several ways we can attempt to create a prototype to satisfy our use-case. Sequence-to-sequence (Seq2Seq) models offer several advantages over earlier machine translation approaches, particularly in handling complex language patterns with longer-range dependencies. Traditional methods, such as rule-based and statistical machine translation, often rely on predefined rules or phrase-based probabilities, which limit their ability to capture the true meaning of text prescribed in different contexts. Seq2Seq models, on the other hand, use deep learning, specifically **encoder-decoder** architectures, to process entire input sequences and generate corresponding output sequences dynamically, for this reason, this is the chosen architecture for modelling our **maths comprehension teaching support assistant**.

**Choice of Model**

One of the key benefits of Seq2Seq models are their ability to be context aware. Instead of translating words or phrases independently, these model process entire sequences, allowing them to understand sentence structures idioms, and syntactic dependencies. Additionally, attention mechanisms can be employed to enhance Seq2Seq models by ensuring that the decoder selectively focuses on relevant parts of the input at each step, preventing loss of important information, especially in longer sequences.

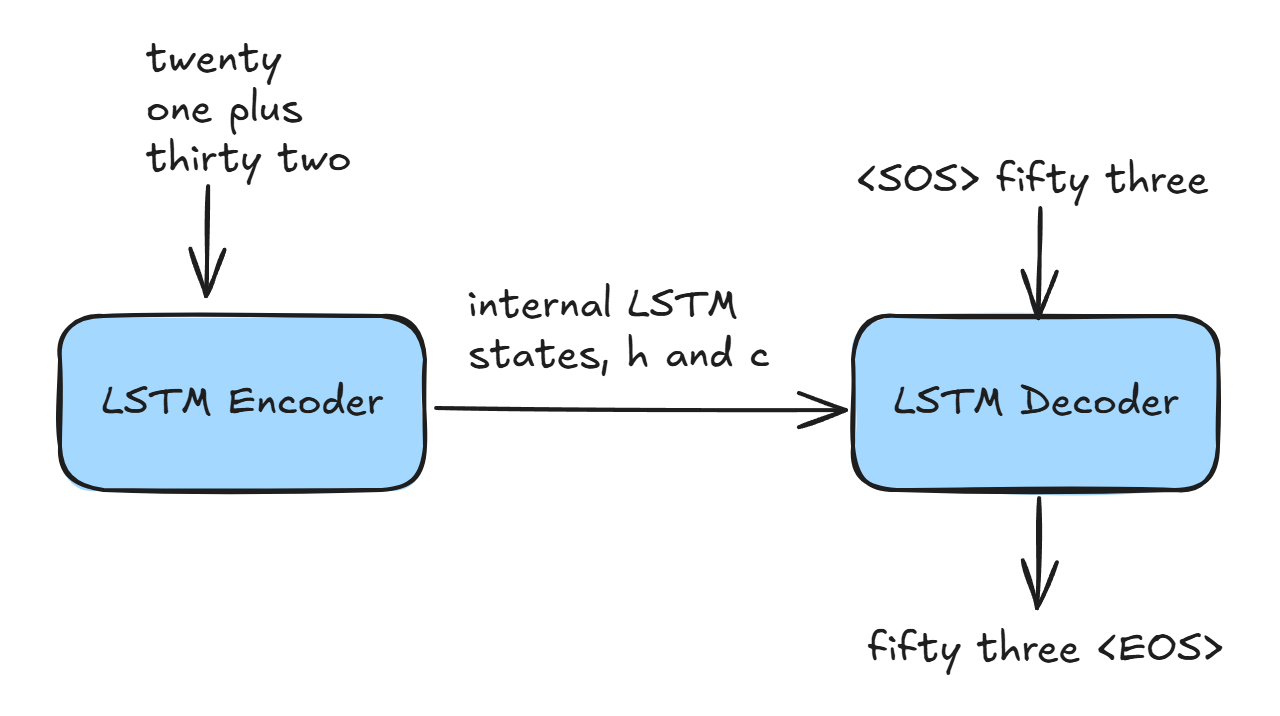
Compared to statistical machine translation techniques, Seq2Seq models produce more fluent and natural translations, as they can learn from larger datasets, rather than relying on rigid phrase-based mappings. Moreover, the flexibility of deep learning architectures enables Seq2Seq models to generalize better to unseen language patterns, adapting well across various domains without manual rule adjustments. So, while Seq2Seq models require more computational resources and larger datasets, their ability to provide accurate, context-aware, and dynamic translations makes it our architecture of choice when modelling our **maths comprehension teaching support assistant**.

**Model Overview**

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 this neural translation machine architecture. We will leverage the power of LSTM models as the basis of our ***encoder-decoder*** setup 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.

As mentioned, our prescribed approach adopts an *encoder-decoder* model 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 the ‘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.

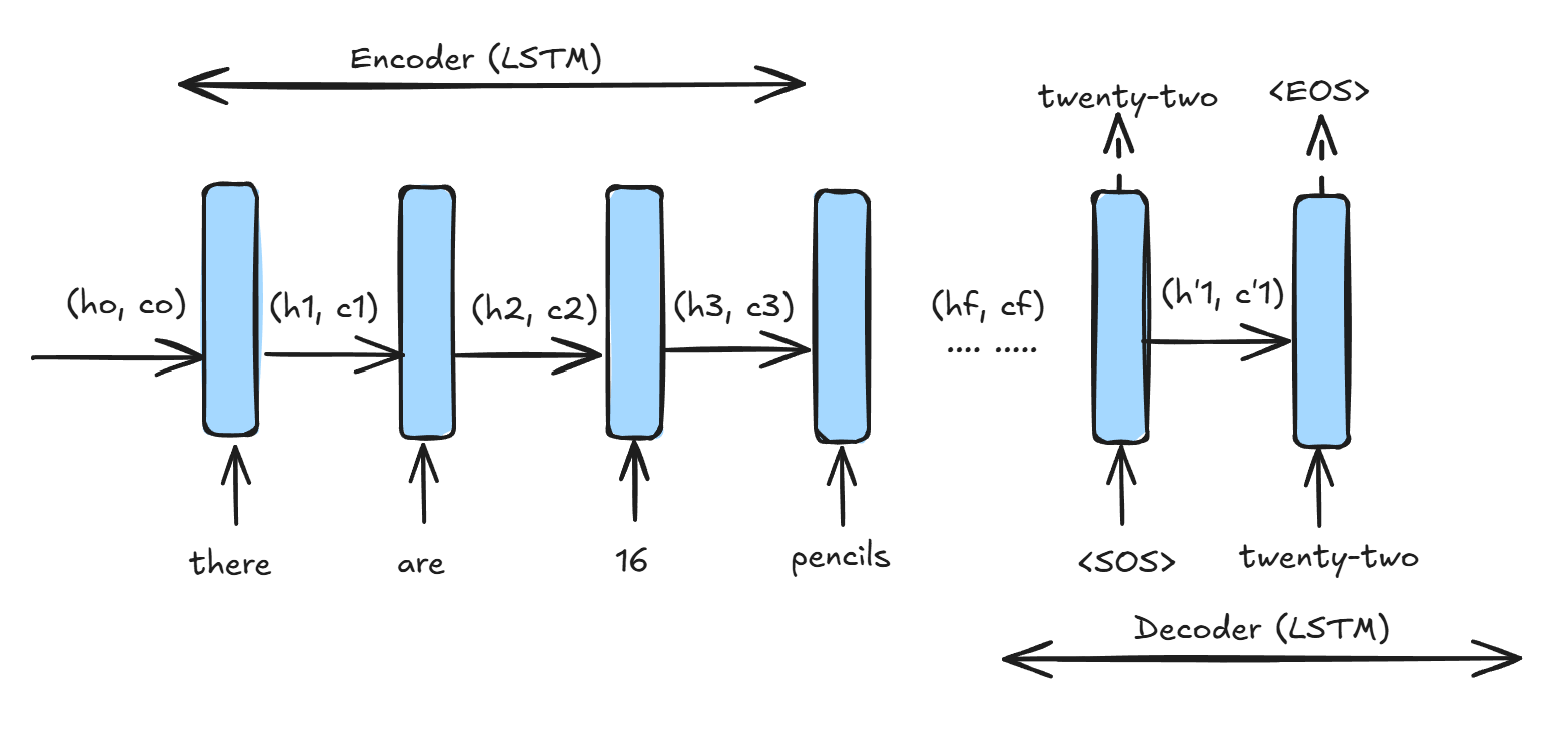
This means the training process for our math assistant can be thought of as building a language model for the target language, in our case the math answers represented by the decoder, conditioned on the hidden states of the encoder that represent the source language, in our case the math questions. Put simply, our seq2seq model is trained using supervised learning. In this approach, our model learns to map the question sequence to our expected answer output sequence based on labelled training data, with the model trained to minimize the difference between its predicted answer output and the actual labelled answer output in the dataset.

**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.

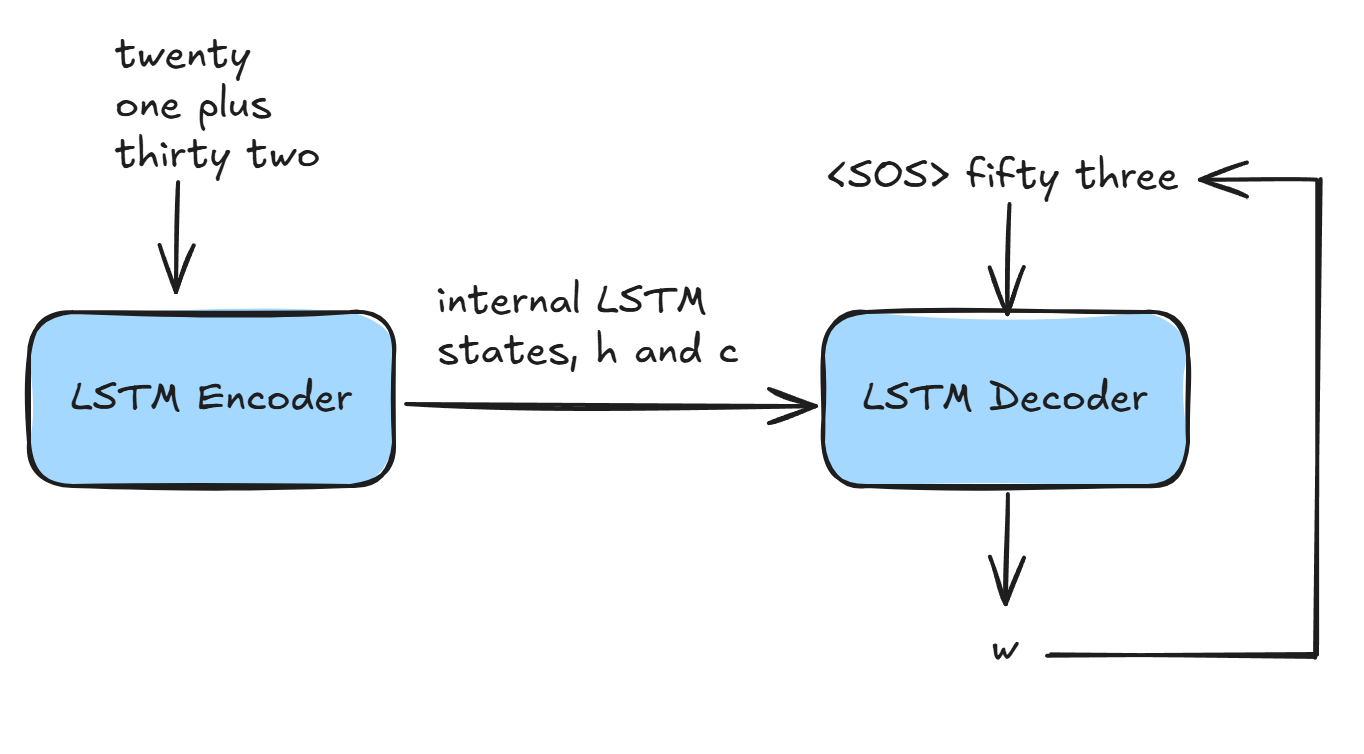
To help our model capture dependencies between words more effectively, we employ an attention mechanism whereby our model learns to assign attention weights dynamically based on our question-answer sequences, refining its ability to focus on relevant tokens and contextual information. Specifically, we utilise a ‘***Bahdanau attention mechanism*’**, which allows our decoder to dynamically focus on different parts of the question sequence at each decoding step, rather than relying on a single compressed representation. The mechanism begins with the encoder, which processes the input sequence and produces a series of hidden state vectors, each representing a word in the input. At each decoding step, an ***alignment score*** is computed, determining the relevance of each encoder hidden state to the current word being generated. This score is calculated using an attention function that takes both the previous decoder hidden state and each encoder hidden state into account. Once these scores are obtained, they undergo ***softmax-normalization***, converting them into probabilities that sum to one. These probabilities dictate the level of attention each input word receives. Using these attention weights, the model computes a ***context vector***, a weighted sum of the encoder’s hidden states. This context vector dynamically adapts based on the decoder’s needs, ensuring that the most relevant parts of our question sequence contribute to generating each answer token. The decoder then combines this context vector with its hidden state to predict the next word, repeating the process for every output. Through this method, ‘***Bahdanau attention***’ should help improve our answer predictions by selectively attending to different parts of the question input. As discussed in the section, ‘***Data Description***’ we use attention visualisation to evaluate the effectiveness of this attention.

We also employ a common technique known as the ‘***dropout rate*’** which refers to the proportion of neurons that are randomly ‘dropped’ (set to zero) during training in each layer of our encoder-decoder models. This technique helps prevent **overfitting** by introducing randomness into the learning process, forcing our model to develop more robust feature representations rather than relying on specific neurons. The initial dropout rate is set to **0.1**, meaning 10% of neurons are temporarily ignored during training. A lower dropout rate is configured in the first instance meaning fewer neurons are dropped, which may not prevent overfitting effectively. However, a higher rate may remove too much information, potentially leading to underfitting, hence the reliance on hyperparameter tuning. It’s important to note that during inference (when making predictions), dropout is turned off, allowing the full network to be used for more stable and make accurate predictions.

With our training overview 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 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,

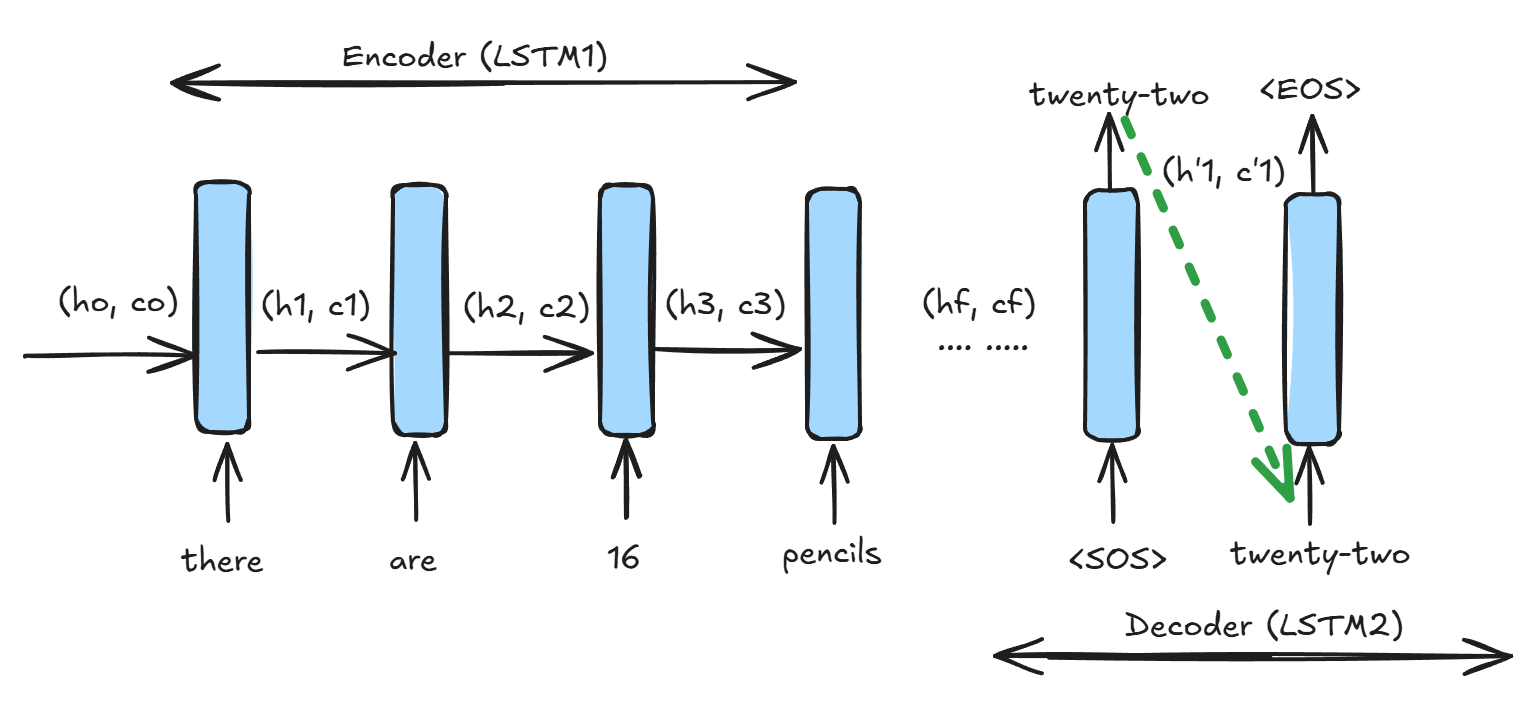
Question: [**6**,**7**,**16**,**4**,**15**,**14**,**5**,**8**,**17**,**4**,**15**,**9**,**5**,**12**,**13**,**3**,**4**,**7**,**11**,**2**]

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 our 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).

**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. 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 approach, 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.

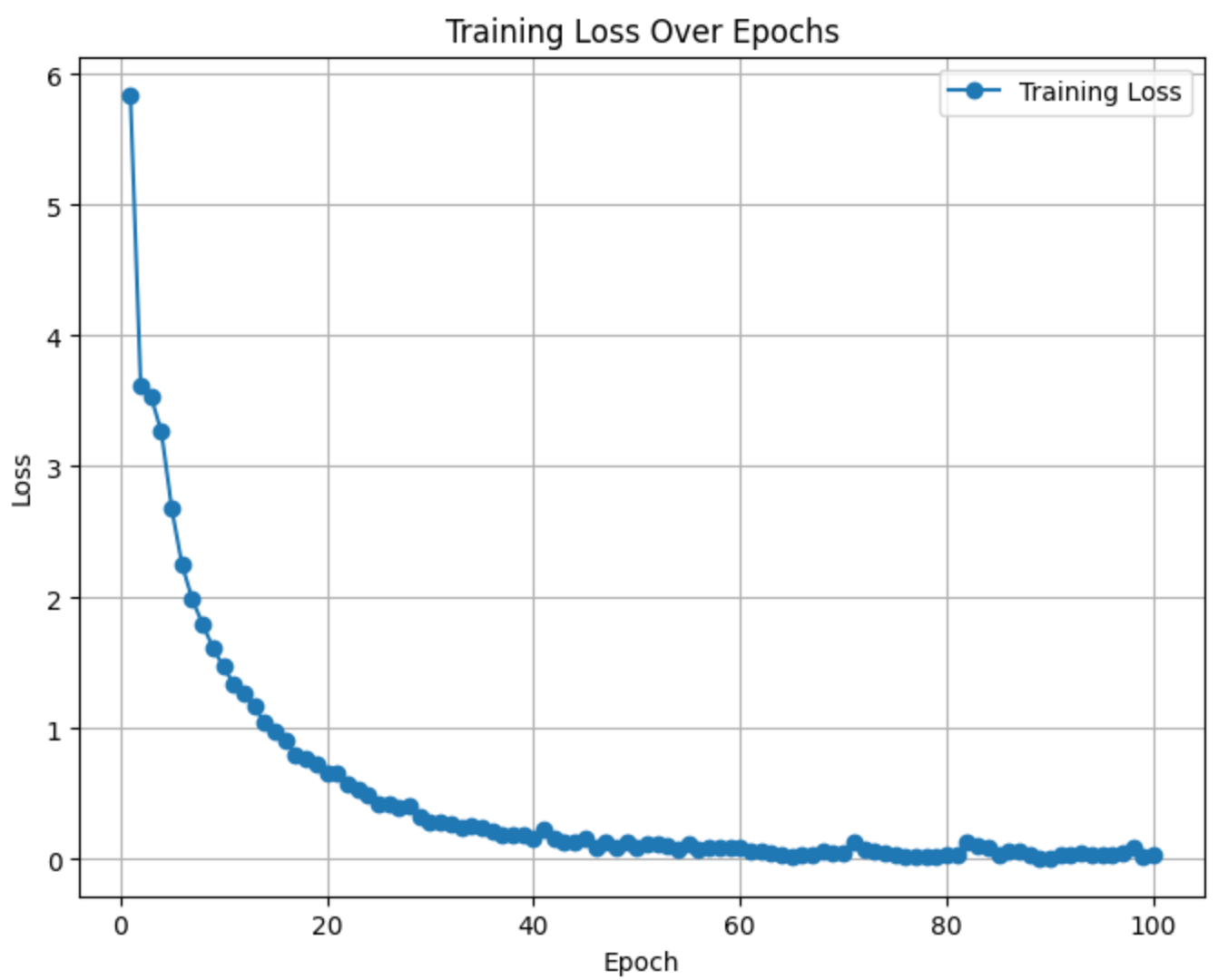
However, it is important to note that this approach of dynamic vocabulary expansion comes with potential challenges, such as inconsistent indexing across different training sessions, making it harder to ensure reproducibility which is certainly an issue that must be addressed if further development is 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.

**Initial Model Training and Evaluation**

For our training run, we initialise the encoder and decoder models with specific parameters; the vocabulary size determines the input and output dimensions, while the hidden size, set to 128, defines the internal (hidden) representation of our data. The loss function is defined using ***CrossEntropyLoss***, which is crucial for optimizing our model during training. By setting ***ignore\_index*** to exclude padding tokens from the loss calculation, the function ensures that irrelevant tokens don't distort the learning process. Additionally, separate Adam optimizers are created for the encoder and decoder, both with a learning rate of ***0.001***, with this optimization approach balances computational efficiency with effective parameter updates during training. To allow interpretability of our model’s attention during training, an empty list (***attention\_matrix***) is initialized to store attention weights produced by the decoder. These weights highlight the parts of the input sequence the decoder focuses on, allowing for deeper insights into the model's decision-making process.

We configure the training loop run for 100 complete iterations through the dataset by assigning the variable ***num\_epochs*** to 100. This should allow our model sufficient time to learn the relevant question-answer patterns and achieve optimization. The training loop processes batches of data, and ‘***teacher forcing***’ is applied, meaning the decoder's next input is the actual target token from the training data rather than its previous output, helping the model learn faster. Loss is calculated only for active sequences, excluding padding tokens, and backpropagation is used to update the model's parameters. After each epoch, the average loss is recorded, and attention weights from the decoder are stored for visualization, providing insights into what the model focuses on. At the end of training, the loss trend is plotted as follows,



Initially, the graph shows a steep decline in loss from approximately 6 down to below 1 within the first 20 epochs. This indicates that the model is learning rapidly during the early stages of training. After this, the loss continues to decrease more gradually and stabilizes around 0.2 for the remaining epochs. This stabilization implies that the model is converging and has reached a point wherein further training yields minimal improvement.

Overall, the graph reflects a well-trained model, where the loss consistently decreases and stabilizes, indicating effective learning and optimization, given the model exhibits a significant drop in loss during early epochs (indicating rapid learning) and then reaches a plateau as the model converges and further improvements become minimal. However, to fully evaluate the model's quality, it is essential to consider other aspects such as attention through visualisation, since attention helps ensure that the decoder is relying on relevant information in the question input sequence, which can lead to more accurate answer predictions.

Using attention maps for visualisation, we can attempt to verify if the model is focusing on logical patterns, making it easier to identify issues like overfitting or poor generalization, as follows,

A graph of a graph

AI-generated content may be incorrect.

If we consider Fig 1., we can see attention is focussed on the bigram “***are there***” which does appear to have a relatively high frequency in our previously generated bigram frequency graph, and perhaps unsurprisingly the trained model predicts the correct answer. The same is true if we consider Fig 2. With a correct prediction made when attention is well focussed on a high frequency bigram, “***sold in***” in this case. However, there are several cases which fail to apply the correct attention and suffer with unreliable predictions as a result, such as that seen in Fig 3. Attention appears to focus on the bigram “***muffins and***” which does not appear in our frequency graph, so we see a poor predicted answer.

It is clear from our selection of generated heatmaps, attention is reasonably focussed on bigrams such as “***are there***” and “***sold in***” which correlate with the bigrams identified in the section ‘***Bigram (two-word phrase) frequency analysis***’,but attention does stray which results in inaccurate predictions.

For further evaluation, we look at the ‘***Edit distance***’, also known as ‘***Levenshtein distance***’, to assess the number of operations required to transform our question into the expected answer by essentially measuring how different the predicted output is from the expected ground truth. In our dataset, each question has a predicted answer and a ground truth answer, and the edit distance evaluates how similar they are. The results from our small test sample are as follows,

A screenshot of a computer program

AI-generated content may be incorrect.

The edit distance results indicate strong model performance, given the majority of predictions align perfectly with the ground truth, leading to an average edit distance of **0.22**. The fact that most test cases have an edit distance of **0** indicates that the model correctly understands and processes arithmetic-based problems. However, minor discrepancies do occur in scenarios involving negative numbers, where the model appears to miscalculate by a few units. Specifically, predicting *"****twenty****"* instead of *"****minus-twenty****"* suggests that the model may struggle when recognizing subtraction results in negative values. Similarly, the misprediction of *"****minus-three****"* instead of *"****minus-two****"* may point to slight calculation errors in handling losses. Such minor inaccuracies suggest the model may require additional training on examples involving subtraction to improve robustness. In this respect, we could adjust the training data distribution to include more subtraction problems. We could also employ hyperparameter tuning to help refine the loss function in order to emphasize error sensitivity when approaching such problems.

Overall, the model performs well but there may be further room for improvement using hyperparameter tuning and other optimisation techniques, together with more rigorous evaluation techniques we can apply.

**Model Refinements, Tuning and Evaluation**

Although evaluation showed our model performed reasonably well, there are a number of operations we can perform in attempt to fine-tune our model for improved performance,

1. ***Positional encodings*** - although LSTMs process data step by step, each token’s placement in the sequence naturally influences its representation. However, explicit positional encodings can still be beneficial;
2. ***K-Fold cross validation*** - used to evaluate our model’s performance while minimising bias and variance, whilst reducing overfitting and maximising data utilisation.
3. ***Hyperparameter tuning*** – utilising a popular third-party utility (***Optuna***) to discover the optimal settings for our model in an attempt to maximise its performance.

**Positional encodings**

Although we recognise our LSTM-based encoder positional information is naturally embedded within the architecture, as the model processes sequences step-by-step, thus preserving order inherently, further incorporation of positional encodings offer the potential to enhance our model’s ability to capture order dependencies which may be characterised by our math comprehension dataset. We do this through explicit positional encodings to improve our model’s comprehension of long-range dependencies and to provide added structure to our question-answer word embeddings before they are fed into the recurrent network. By incorporating positional encodings, the encoder can differentiate between word sequences like *"****total in****"* and *"****in total****"* even though the individual words remain the same. The revised encoder model incorporates positional encoding by explicitly adding positional information to input embeddings before passing them through the LSTM. This is achieved using a separate embedding layer for positional encodings, which assigns a unique vector to each position in the sequence.

**Hyperparameter tuning**

We also attempt to improve the hyperparameter established for our model since these parameters are typically fixed before our training regime begins and cannot be learned from our dataset, unlike the model parameters, which are constantly adjusting during training. Examples of hyperparameters include learning rate, number of hidden layers in a neural network, and the number of trees in a random forest. Our model configuration incorporates several essential hyperparameters that influence model performance and training efficiency. The ***input\_size*** and ***output\_size*** are set to the vocabulary size, determining the range of token embeddings handled by the encoder and decoder. The ***hidden\_size*** defines the dimensional space for the respective encoder-decoder LSTM’s hidden states, affecting how well our model captures question-answer dependencies. The learning rate controls the update step for weight adjustments, ensuring a balance between stable convergence and learning speed. Additionally, we include an ***ignore\_index*** argument in our ***CrossEntropyLoss*** function which helps the model ignore padding tokens, thus preventing them from negatively impacting gradient updates.

In order to automatically tune our hyperparameters to their best fit, we leverage the ***Optuna*** open-source hyperparameter optimisation framework, which is a powerful tool for hyperparameter tuning, providing an efficient way to optimize machine learning models without relying on exhaustive manual searches. ***Optuna*** was chosen because unlike traditional grid searches, which systematically test all possible hyperparameter combinations, or random searches, which selects values without guidance, ***Optuna*** applies Bayesian optimisation to intelligently explore the hyperparameter space, reducing computational time while improving model performance. If we consider the results generated,

A screen shot of a computer screen

AI-generated content may be incorrect.

It is clear how different hyperparameter combinations impact model performance. The trials explored variations in hidden layer sizes, learning rates, and dropout rates, aiming to find the most effective configuration. The best trial achieved a value of **4.1879**, with a hidden size of **212**, a relatively high learning rate of **0.0097**, and a dropout rate of **0.4932**. This suggests that a moderate hidden size, a higher learning rate, and a significant dropout rate contribute to better model performance for our model setup.

A diagram of a plot

AI-generated content may be incorrect.

Looking at broader trends, trials with extremely low learning rates (e.g., **Trial 2 and Trial 5**) performed poorly, reinforcing the importance of a sufficiently high learning rate for effective optimization. The dropout rate seems to have a balancing effect—while too high a dropout rate hindering learning, whereas a moderate-to-high rate (around **0.49**) proved beneficial. The hidden size didn’t show a clear linear pattern, but the best trials tended to have values in the 200–400 range.

**K-Fold Cross Validation**

K-fold cross-validation is a popular technique in machine learning to evaluate model performance while reducing bias and variance. As such, it is a natural choice to our sequence-to-sequence model. By doing so, we are ensuring that training and validation are conducted on multiple subsets of our training dataset rather than relying on a single data split. This approach enhances generalisation and helps identify the best model configuration by testing it across multiple partitions. We utilise the ***K-Fold*** functionality provided by the Scikit-learn library to divide the dataset into 5 k-fold subsets. For each fold, one subset is used for validation, while the remaining k-1 subsets are used for training. By shuffling the dataset and setting a fixed random state of 42 (thanks Douglas Adams), the data distribution should remain consistent across experiments, preventing unintended biases from influencing the results. The model is trained sequentially on each fold, ensuring that all data points serve as validation samples at least once.

Before training begins on a new fold, we ensure the model’s weights are reinitialised, allowing each iteration to be refreshed without being influenced by previous folds. The optimizer is initialized with a learning rate of 0.001, in an attempt to balance convergence speed and model stability. Training follows a ‘***teacher forcing*’** approach, where the decoder is guided using actual target tokens to improve sequence predictions. During each epoch, the model processes batches of data and optimizes its parameters using ‘***cross-entropy loss***’, where padding tokens are ignored to prevent unnecessary penalization.

After completing training on each fold, the model is evaluated on the validation set using an evaluation function that tracks the average validation loss. The validation losses for all folds are recorded, allowing comparison of fold performance. The best-performing encoder-decoder pair is selected based on the lowest validation loss, ensuring that the final trained model is the most generalizable across different data splits.

**Bleu score**

To evaluate our results, we look to leverage a process referred to as **BLEU (Bilingual Evaluation Understudy) score** is a widely used metric in natural language processing, the goal here being to measure how well our model performs in model in generating responses to math word problems. BLEU works by comparing n-grams (short sequences of words) between the predicted answer sequences and the reference (ground-truth) answer sequences. BLEU quantifies similarity based on n-gram precision, checking how many word sequences (single words, bigrams, trigrams, etc.) in the generated text also appear in the reference text. The score ranges from **0** to **1**, with higher values indicating greater overlap and better translation accuracy. We calculate BLEU scores batch-wise and for the entire dataset, and since our approach tokenizes sequences before processing, we must translate the predicted token sequences back into word form before computing BLEU, ensuring a fair comparison. We also apply corpus-level **BLEU**, aggregating predictions across all batches rather than evaluating individual sentences separately. An important aspect of BLEU in our case is how it handles the short answers in our math problems. Since our expected mathematical responses tend to be brief, applying brevity penalty adjustments ensures the model isn’t unfairly rewarded for producing overly short outputs. If we consider the generated results,

A screen shot of a graph

AI-generated content may be incorrect.

A BLEU score of **100.0** (or effectively **100%**) indicates **perfect match** between our models generated answer outputs and the reference question-answer sequences. If we consider this result in a superficial way, this means that our math comprehension solver is producing identical outputs compared to the expected answers, without deviations in structure or wording. These seems dubious, and while a high BLEU score is certainly desirable, a perfect score could suggest that our model is memorising the training data rather than truly understanding problem-solving patterns.

**Results, Conclusions and Recommendations**

Our **maths comprehension teaching support assistant,** sequence-to-sequence (encoder-decoder) model has demonstrated strong potential in understanding and processing the comprehension type mathematical questions asked of it. By leveraging an attention mechanism, the model was able to effectively identify patterns in question-answer steps, achieving a good level of accuracy throughout these comprehension tasks. The optimization process using ***Optuna*** helped refine the hyperparameter configurations, such as learning rate, hidden layer configurations, and dropout settings, again resulting in improved performance. However, challenges remain in handling highly ambiguous problems, and intricate mathematical expressions that require deep contextual understanding. This may be partly due to how we constrained the training data in size and contextual scope (i.e. focus on KS1 style questions only), so whilst the current setup achieves satisfactory results, further enhancements could improve the model’s robustness and interpretability. We also observed a potentially inaccurately high BLEU score which will certainly lead us into further tuning and evaluation processes.

In this respect, further hyperparameter optimisation could be conducted. The best trial in the study showed a promising learning rate of **0.0097**, and fine-tuning around this value could refine efficiency. The hidden size parameter requires careful evaluation, as variations between **200** and **400** have shown differing outcomes, making it necessary to experiment with slightly larger and smaller values to determine optimal complexity. Additionally, dropout rates around **0.49** have shown success, but slight refinements may further enhance generalization without excessive information loss.

As discussed, the quality and diversity of training data may well play a crucial role in enhancing our model’s performance. Expanding the dataset to include a wider range of mathematical problems, including multi-step logical reasoning, could improve adaptability. Likewise, increasing the scope of synthetic problem augmentation may challenge the model further by providing a broader set of examples for learning. Furthermore, implementing strategies for error analysis could help identify patterns in incorrect predictions, enabling targeted improvements in model training. If the results of applying such recommendations results in still limited model performance, other architectures exist in this AI space which could be exploited.

Indeed, exploring more enhanced model architectures could provide substantial improvements such as ‘***Transformer-based***’ approaches, known for their effectiveness in sequence-to-sequence tasks, could help refine the reasoning process and strengthen generalization capabilities. Additionally, reinforcement learning techniques may enhance the model’s ability to follow structured problem-solving steps, improving interpretability. Introducing mechanisms that provide explanations for each solution step would also benefit users by ensuring transparency in how mathematical problems are solved.

Whichever approach we decide to pursue, we need to ensure stability, reliability, and accuracy before the model can be properly harnessed in a production ready application. In this respect, the model must undergo robustness testing, perhaps including adversarial examples and stress-testing against varied mathematical structures. By addressing these areas, our encoder-decoder could be transformed into a more effective and adaptable tool for mathematical comprehension problem-solving, catering to a wide range of problem complexities. Indeed, if the longer-term aim is to extend our models ability to adapt to problems across the Key-Stage STEM spectrum then this quality of robustness must be guaranteed.

**References**

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

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

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

***NLP From Scratch: Translation with a Sequence to Sequence Network and Attention***

Pytorch: Created On: Mar 24, 2017 | Last Updated: Oct 21, 2024 | Last Verified: Nov 05, 2024

<https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html>

***10.7. Sequence-to-Sequence Learning for Machine Translation***

Dive into Deep Learning

<https://d2l.ai/chapter_recurrent-modern/seq2seq.html>

**[Hands-On] Implementing a Simple Mathematical Calculator using Sequence-to-Sequence Learning**

Hugman Sangkeun Jung on March 20th 2024

<https://medium.com/@hugmanskj/hands-on-implementing-a-simple-mathematical-calculator-using-sequence-to-sequence-learning-85b742082c72>

**Hand-On Machine Learning with Scikit-Learn, Keras & Tensorflow – 2nd Edition**

Aurelien Geron