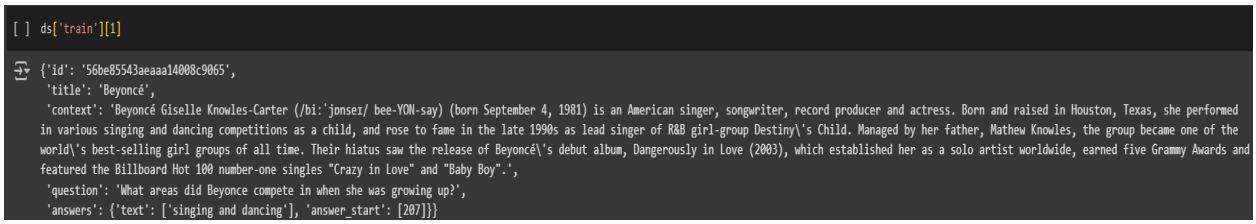
NLP Milestone 2

Dataset and pre-processing

The dataset we chose was SQuAD (Stanford Question Answering Dataset):

- This dataset had around 140000 rows
- We only selected 20000 rows out of the dataset. The rows were shuffled and then split into 17000 for training, 1500 for validation and 1500 for testing.
- Each row had a context, question and an answer which had the start index and the text of the answer. Here is an example of how a row looked:



The preprocessing done was:

- The irrelevant fields like id and title were dropped.
- We extracted the **context**, **questions** and **answers** as separate arrays. For the answers we only took the text and dropped the answer_start.
- We removed all the occurrences of **'\n'** in all the text.
- We then calculated the maximum word count for the context, question and answer and we saved them to be used later on.
- We added **special Tokens for sequence control**. Each answer is prepended with a **<start>** token and appended with an **<end>** token. These modified sequences are split into two parts:
    - `answers_in`: fed as input to the decoder during training.
    - `answers_out`: used as the target sequence that the model is trained to predict.
- All text data including contexts, questions, and both versions of answers are **tokenized** using a shared **`Tokenizer`**. This ensures a consistent vocabulary across all components of the input and output sequences. Tokenizer is fitted on the complete corpus: contexts, questions, and answer sequences. The sequences are split into words and each word is converted to a unique index. Filtering is disabled (`filters=''`) to preserve all punctuation and special tokens like `<start>` and `<end>`.

- We padded all the sequences to have a uniform length. **Contexts** and **questions** are padded to **`max_context_length`** and **`max_question_length.`** `Decoder inputs (`**`answers_in`**`) and outputs (`**`answers_out`**`) are padded to` **`max_answer_length`** `+ 1`

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Concatenate, Dropout
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# add <start> at the start of every answer
answers_in = ["<start> " + answer for answer in train['answers']]
answers_out = [answer + " <end>" for answer in train['answers']]
contexts = train['context']
questions = train['question']


all_texts = contexts + questions + answers_in + answers_out
tokenizer = Tokenizer(filters='')
tokenizer.fit_on_texts(all_texts)


context_seq = pad_sequences(tokenizer.texts_to_sequences(contexts), maxlen=max_context_length, padding='post')
question_seq = pad_sequences(tokenizer.texts_to_sequences(questions), maxlen=max_question_length, padding='post')
answer_in_seq = pad_sequences(tokenizer.texts_to_sequences(answers_in), maxlen=max_answer_length+1, padding='post')
answer_out_seq = pad_sequences(tokenizer.texts_to_sequences(answers_out), maxlen=max_answer_length+1, padding='post')
```

Methodology

This project implements a context-question answering system using an encoder-decoder architecture based on LSTM networks. The goal is to generate accurate natural language answers given a context passage and a related question. The overall methodology consists of data preprocessing, model architecture design, training strategy, and inference setup.

1. Data Representation

The input for our model consists of three textual components:

- **Context**: A passage of text that contains the information required to answer a question.
- **Question**: A question about the context.
- **Answer**: A corresponding answer, generated in natural language.

The max length for each component that was calculated after tokenization and padding is used to specify the input length:

- `max_context_length` for context input.
- `max_question_length` for question input.
- `max_answer_length + 1` for decoder input (with a start token).

The vocabulary size is denoted by `VOCAB_SIZE`, and the padding token is assumed to be `0`.

2. Model Architecture

2.1 Encoder

The encoder processes both the context and the question independently using shared embeddings followed by separate LSTM layers. The key components are:

- **Embedding Layer**: All inputs are embedded using a shared `Embedding(VOCAB_SIZE, 256, mask_zero=True)` layer to convert token indices into dense vectors of dimension 256.
- **Dropout Layer**: A dropout of 30% is applied to both context and question embeddings to prevent overfitting.
- **Context Encoder**: A unidirectional LSTM with 128 hidden units processes the context sequence and outputs the final hidden state and cell state: `(state_h1, state_c1)`.

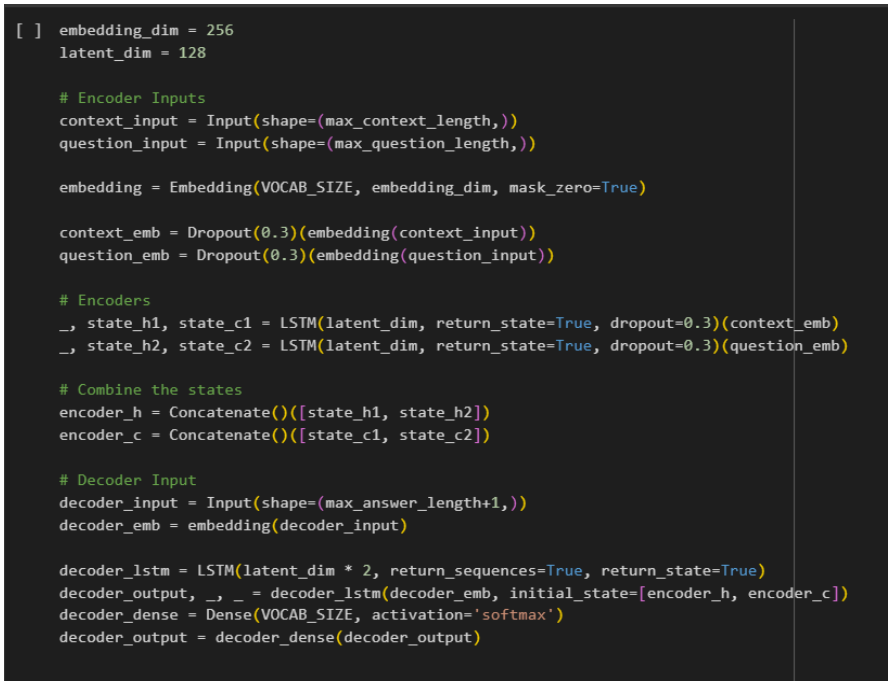
- **Question Encoder**: A separate LSTM processes the question sequence similarly to produce `(state_h2, state_c2)`.
- **State Combination**: The final hidden and cell states from both the context and question encoders are concatenated to form the initial states for the decoder.

This design allows the decoder to condition its generation on both the context and the question.

2.2 Decoder

- **Input**: The decoder receives the answer sequence as input during training, with an embedding layer identical to the encoder.
- **LSTM Layer**: A single LSTM with `latent_dim * 2` (i.e., 256 units) is used to match the concatenated encoder state dimensions.
- **Dense Output Layer**: The LSTM output is passed through a Dense layer with softmax activation to produce token-level probability distributions over the vocabulary.

The decoder is trained to predict the next token in the answer sequence.



```
[ ]  embedding_dim = 256
     latent_dim = 128

     # Encoder Inputs
     context_input = Input(shape=(max_context_length,))
     question_input = Input(shape=(max_question_length,))

     embedding = Embedding(VOCAB_SIZE, embedding_dim, mask_zero=True)

     context_emb = Dropout(0.3)(embedding(context_input))
     question_emb = Dropout(0.3)(embedding(question_input))

     # Encoders
     _, state_h1, state_c1 = LSTM(latent_dim, return_state=True, dropout=0.3)(context_emb)
     _, state_h2, state_c2 = LSTM(latent_dim, return_state=True, dropout=0.3)(question_emb)

     # Combine the states
     encoder_h = Concatenate()([state_h1, state_h2])
     encoder_c = Concatenate()([state_c1, state_c2])

     # Decoder Input
     decoder_input = Input(shape=(max_answer_length+1,))
     decoder_emb = embedding(decoder_input)

     decoder_lstm = LSTM(latent_dim * 2, return_sequences=True, return_state=True)
     decoder_output, _, _ = decoder_lstm(decoder_emb, initial_state=[encoder_h, encoder_c])
     decoder_dense = Dense(VOCAB_SIZE, activation='softmax')
     decoder_output = decoder_dense(decoder_output)
```

3. Inference Setup

During inference, a separate architecture is used to allow step-by-step generation:

- **Encoder Inference Model**: A model is created to output the combined encoder states (`encoder_h`, `encoder_c`) given the context and question inputs.
- **Decoder Inference Model**:
    - Takes a single input token (1 time step) and the previous decoder states.
    - Outputs the next token probabilities and updated states.
    - This setup enables generating one token at a time and feeding it back into the decoder for the next step.

```
[ ]  # Encoder model
     encoder_model = Model([context_input, question_input], [encoder_h, encoder_c])

     # Decoder inference setup
     decoder_state_input_h = Input(shape=(latent_dim * 2,))
     decoder_state_input_c = Input(shape=(latent_dim * 2,))
     decoder_inputs_single = Input(shape=(1,))
     decoder_inputs_single_emb = embedding(decoder_inputs_single)

     decoder_outputs, state_h, state_c = decoder_lstm(
         decoder_inputs_single_emb,
         initial_state=[decoder_state_input_h, decoder_state_input_c]
     )
     decoder_outputs = decoder_dense(decoder_outputs)

     decoder_model = Model(
         [decoder_inputs_single, decoder_state_input_h, decoder_state_input_c],
         [decoder_outputs, state_h, state_c]
     )
```

4. Training Strategy

- **Loss Function**: Categorical cross-entropy which is used in multi-class classification is used as the loss function. This loss function is used as at each time-step the model is predicting a range of possibilities of different words (classes) which we choose the best from.
- **Optimizer**: The model is trained using the Adam optimizer.
- **Regularization**: Dropout is applied at multiple stages to mitigate overfitting.
- **Evaluation Metrics**:

- **F1-score**: F1 score is a machine learning evaluation metric that evaluates the model by combining the precision and recall scores of a model. The accuracy metric computes how many times a model made a correct prediction across the entire dataset. The accuracy is not very good when evaluating nlp models as the output generated text will not be exactly the same as the target. Precision measures how many of the “positive” predictions made by the model were correct. Recall measures how many of the positive class samples present in the dataset were correctly identified by the model. F1 uses both using the following equation:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{F1 Score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}}$$

$$\text{Recall} = \frac{TP}{TP + FN} \qquad = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Rouge:** The ROUGE metrics are commonly used for evaluating automatic text generation tasks like summarization or translation by comparing the generated text (candidate) to a reference text (usually human-written).

Post-processing

After obtaining the context and question representations from the encoder, the decoder generates the answer token by token using a greedy decoding approach.

First we tokenize and pad the context and question to the max length we calculated during pre processing.

Then we pass them through the encoder to get the cell state and the hidden state to pass to the decoder.

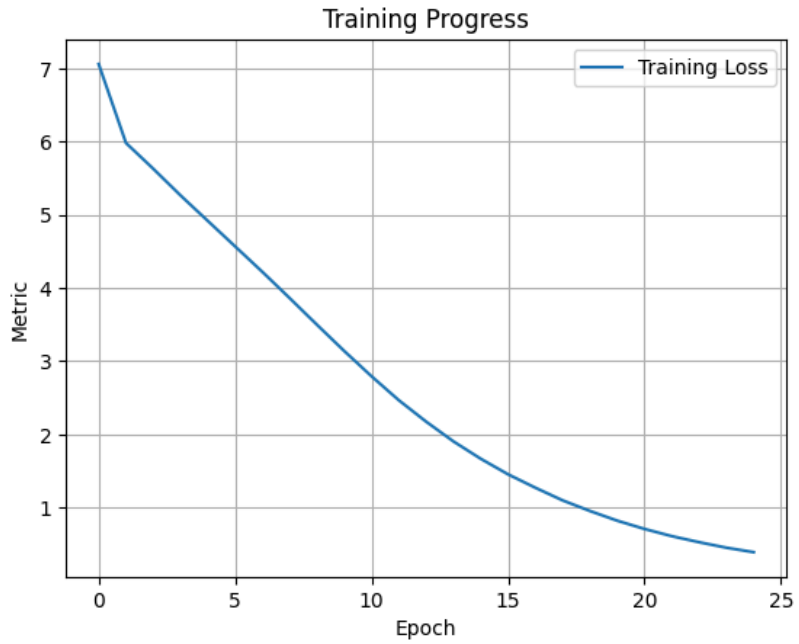
We give the decoder the start token as the first word and then the decoder decides which word in the vocabulary is most likely to come next, this process is repeated for a max number of 10 steps or until the end token is generated.

Then all these words are appended and returned as a string to the user.

```python
def generate_answer(context_text, question_text):
    # Encode context and question
    context_seq = pad_sequences(tokenizer.texts_to_sequences([context_text]), maxlen=max_context_length, padding='post')
    question_seq = pad_sequences(tokenizer.texts_to_sequences([question_text]), maxlen=max_question_length, padding='post')
    state_h, state_c = encoder_model.predict([context_seq, question_seq])

    # Start decoding
    target_seq = np.array([[tokenizer.word_index['<start>']]])
    stop_condition = False
    decoded_sentence = []

    for _ in range(10):  # Max answer length
        output_tokens, h, c = decoder_model.predict([target_seq, state_h, state_c])
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        #print('index:', sampled_token_index)
        sampled_word = tokenizer.index_word.get(sampled_token_index, '')
        #print('word:', sampled_word)

        if sampled_word == '<end>' or sampled_word == '':
            break

        decoded_sentence.append(sampled_word)

        target_seq = np.array([[sampled_token_index]])
        state_h, state_c = h, c

    return ' '.join(decoded_sentence)
```

Results

Loss plot:



Training f1-score : 0.9014292453228491

Training rouge:

- **rouge1**: 0.9060049023573746,
- **rouge2**: 0.5930859823801929,
- **rougeL**: 0.9055293126080426,
- **rougeLsum**: 0.9052159803743351

Validation f1-score: 0.04603139920097312

Validation rogue:

- **rouge1**: 0.05003525723997221,
- **rouge2**: 0.017203400858190325,
- **rougeL**: 0.049757762425898265,
- **rougeLsum**: 0.05012946180726184

Testing f1-score: 0.0507737634786304

Testing rogue:

- **rouge1**: 0.05700964168535229,
- **rouge2**: 0.016584071920963183,
- **rougeL**: 0.05705254707936515,
- **rougeLsum**: 0.05669823377940772

Limitations

Limited Dataset Size:
Only **17,000 rows** from the original dataset were used for training. This relatively small subset reduced the model's ability to be able to generalize, particularly when encountering complex or uncommon question-context pairs.

Shallow Model Architecture:
The encoder-decoder model was restricted to just **3 layers**, which is relatively shallow for capturing the complex semantic relationships required for question answering. Deeper architectures could potentially learn more representations and connections and improve overall performance.

No Combination of LSTM and Attention:
Another constraint was that we were **not allowed to combine LSTM with attention** in the same model. This hybrid approach is commonly used in real-world NLP systems and could have significantly enhanced performance.

Suggested Improvements

- Increase the size of the dataset used during training to allow the model to see more vocabulary and be able to generalize more.
- Perform data augmentation to increase the variations of the questions and answers by changing the locations of the answer in the context and asking the same questions in different ways. This will allow the model to generalize more and be exposed to different variations of text.

Experimentation

In addition to the initial generative sequence-to-sequence model using LSTM, an alternative **extractive approach** was explored to improve performance and more directly leverage the structure of the dataset.

This method aimed to **extract the answer span directly from the context** rather than generating it word-by-word. The core idea is that the model predicts the **start and end indices** of the answer within the context.

Model Architecture:

- The **context** and **question** were concatenated with a special `[SEP]` token between them.
- The concatenated sequence was embedded using **BERT embeddings** (without using the full BERT model, only the embeddings).
- The embeddings were then passed through a layer of **multi-headed self-attention**.
- A **residual connection and layer normalization** were applied to stabilize learning and maintain information from the original input.
- The resulting sequence was fed into a **dense layer**, which predicted two probability distributions over the input tokens: one for the **start index** and one for the **end index** of the answer span.

Performance:

- The model achieved approximately **40% ROUGE score on training data**, but only **6–7% on the test set**, indicating a significant overfitting problem.
- After applying some basic regularization techniques to mitigate overfitting (e.g., dropout, fewer layers), the results shifted to around **10% on training** and **8% on testing**, showing slight generalization improvement but still underperforming overall.