Inserting Shakespeare dataset into Tensorflow's Colab

The model in Tensorflow's colab:

The tensorflow colab is based on a Transformer model to be a chatbot, so the model is basically a Transformer that uses self-attention —the ability to attend to different positions of the input sequence to compute a representation of that sequence. Since this model is *designed to be a chatbot* the data given to it should be in a format of two tensors, *inputs and outputs.* Then the inputs should be divided to inputs and decoder inputs.

The files given:

When we first checked out the datasets given, they were two, the Cornell Movie-Dialogs Corpus and the Tiny Shakespeare dataset. Both datasets where to be downloaded from zip files, those zip files are downloaded on our local machine according to the two unchanged colabs. The Cornell Movie-Dialogs Corpus in the Tensorflow colab would be extracted from the zip files through the "tf.keras.utils.get_file" and then "os.path.join" both working together to extract two file, the moves_lines.txt and the movie_conversations.txt. The movies_lines and movie_conversations would then both be used to make two dictionaries, the questions and answers, which we have assumed to be the inputs and outputs respectively.

The problem:

the skahespear dataset is only one file and has been used in the pytorch colab which is a generative transformer meaning it generates words and not a chatbot. The skakespeare dataset wasn't a question and answer it or two characters talking to each other. Another problem was that the data preprocessing in tensorflow relied entirely on the dataset coming in two files as explained above, *movies_lines and movie_conversations* which we dont have in the Shakespeare dataset. Functions like load_conversations(): and tokenize_and_filter() need those two files.

The Approach:

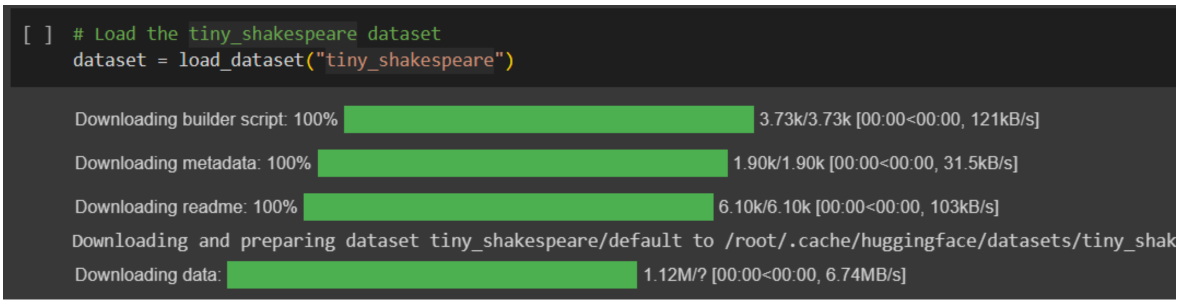
we have thought of a solution, the approach was to make our own preprocessing on the shakespear dataset, more specifically, we thought of splitting the shakespeare dataset into two tensors, the inputs and outputs. tThen we thought of using the "[tf.data.Dataset.from_tensor_slices]" method that creates a TensorFlow `Dataset` from the preprocessed input-output pairs. Below we will explain the approach in more details. To achieve the results, we have decided that it's way easier to use a dataset loaded from the dataset library from HuggingFace rather than downloading it as zip files and writing code specifically for extracting files from the zip files. Below will hold more details.

Our new tokenization:

As we have said above, we had to download the dataset from huggingface, which has been down using

```
import datasets
from datasets import load_dataset
# Load the tiny_shakespeare dataset
dataset = load_dataset("tiny_shakespeare")
```

After writing the above code, we succeeded in installing the tiny_shakespeare dataset.



Now the next step was to make our own tokenization which we have done by importing a library called GPT2Tokenizer from Transformers

```
from transformers import GPT2Tokenizer
```

Then we had to initialize the tokenizer using the following code.

```
# Initialize the tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})
```

which has resulted in the following output

```python
# Initialize the tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})
```

```
Downloading (…)olve/main/vocab.json: 100%  1.04M/1.04M [00:00<00:00, 5.23MB/s]
Downloading (…)olve/main/merges.txt: 100%  456k/456k [00:00<00:00, 2.41MB/s]
Downloading (…)lve/main/config.json: 100%  665/665 [00:00<00:00, 48.1kB/s]
Using pad_token, but it is not set yet.
```

Next was to preprocess the data, we have decided to add the preprocessed data which will be a dictionary to a list called preprocessed_data

```python
[ ]  # Preprocess the data
     preprocessed_data = []
```

Using a for loop, we have iterated over the examples in the "train" split of the dataset.

```python
for example in dataset["train"]:
    text = example["text"]
    tokenized_text = tokenizer(text, truncation=True, padding="max_length", max_length=40, return_tensors="pt")
    input_ids = tokenized_text["input_ids"].squeeze()
    attention_mask = tokenized_text["attention_mask"].squeeze()
```

The code retrieves he text from the current example. It uses the "text" key in the example dictionary and holds the text data. Then the code tokenizes the text using the tokenizer object, which is initialized earlier in the code the tokenizer takes the text as input and performs tokenization with the parameters `truncation=True` and `padding="max_length"` ensure that the text is truncated and padded to a maximum length of 40 tokens. The `return_tensors="pt"` parameter indicates that the tokenizer should return PyTorch tensors. Then the code extracts the "input_ids" tensor from the tokenized text and applies the `squeeze()` method to remove any unnecessary dimensions. The last line extracts the "attention_mask" tensor from the tokenized text and applies the `squeeze()` method to remove any unnecessary dimensions.

Now we had to append the extracted dictionaries into the pre processed_data list as follows.

```python
preprocessed_data.append({
        "input_ids": input_ids,
        "attention_mask": attention_mask,
        "text": text
    })
```

This has facilitated the process of splitting which will happen as follows,

```
# Split into input and output pairs
input_pairs = []
output_pairs = []
```

first we have created two empty lists to be appended in, and below we have appended the lists.

```
for example in preprocessed_data:
    input_sequence = example["input_ids"]
    output_sequence = example["input_ids"]  # Exclude the first token (shifted input)

    input_pairs.append(input_sequence)
    output_pairs.append(output_sequence)
```

```
output_pairs
```

```
[tensor([ 5962, 22307,    25,   198,  8421,   356,  5120,   597,  2252,    11,
          3285,   502,  2740,    13,   198,   198,  3237,    25,   198,  5248,
           461,    11,  2740,    13,   198,   198,  5962, 22307,    25,   198,
          1639,   389,   477, 12939,  2138,   284,  4656,   621,   284,  1145])]
```

```
input_pairs
```

```
[tensor([ 5962, 22307,    25,   198,  8421,   356,  5120,   597,  2252,    11,
          3285,   502,  2740,    13,   198,   198,  3237,    25,   198,  5248,
           461,    11,  2740,    13,   198,   198,  5962, 22307,    25,   198,
          1639,   389,   477, 12939,  2138,   284,  4656,   621,   284,  1145])]
```

lastly we have converted the lists to tensors this way we have achieved the goal of having inputs and outputs to be fed into our model.

```
# Convert to tensors
input_tensor = torch.stack(input_pairs)
output_tensor = torch.stack(output_pairs)
```

Finally we have two tensors as input and output to be used in the dataset but we have to first merge them into one dataset with "[tf.data.Dataset.from_tensor_slices]" as follows.

```python
# decoder inputs use the previous target as input
# remove START_TOKEN from targets
dataset = tf.data.Dataset.from_tensor_slices(
    (
        {"inputs": input_tensor, "dec_inputs": output_tensor[:, :-1]},
        {"outputs": output_tensor[:, 1:]},
    )
)
```

The results

The below results show how poorly the training has done because the model didn't even show any sort of words which suggests it didn't learn anything which is not the goal and is not satisfying, **we made a few experiments by changing the hyperparameters** and trying to find out why it didn't work but they all seemed to give the same results which has led us to change our approach and decided to redo the dataset swapping.

Let's test our model!

```python
[ ] predict("Where have you been?")
```

```
'YouYou\n\n\nYou\nYou\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\nYouYouYouYouYouYouYouYouYouYouYouYouYou'
```

```python
[ ] predict("It's a trap")
```

```
'You\nYou\n\n\n\nYou\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\nYouYouYouYouYouYouYouYouYouYouYouYouYou'
```

```python
[ ] # feed the model with its previous output
    sentence = "I am not crazy, my mother had me tested."
    for _ in range(5):
        print(f"Input: {sentence}")
        sentence = predict(sentence)
        print(f"Output: {sentence}\n")
```

```
Input: I am not crazy, my mother had me tested.
Output: You CitizenYou CitizenYou
YouYouYou
```

Approach Two

Again we have installed the Tiny Shakespeare dataset using the load_dataset function from the library dataset. This time we have decided to **use the original tokenization that has been implemented in the original dataset** this means that we have removed our own tokenization and added the original one as shown below.

This also meant we had to find a way to break down the dataset into two, inputs and outputs. We have done so as explained below.

```
import itertools
def preprocess_sentence(sentence):
    sentence=list(itertools.chain(*sentence))
    sentence=" ".join(sentence)
    # creating a space between a word and the punctuation following it
    # eg: "he is a boy." => "he is a boy ."
    sentence = re.sub(r"([?.!,])", r" \1 ", sentence)
    sentence = re.sub(r'[" "]+', " ", sentence)
    # removing contractions
    sentence = re.sub(r"i'm", "i am", sentence)
    sentence = re.sub(r"he's", "he is", sentence)
    sentence = re.sub(r"she's", "she is", sentence)
    sentence = re.sub(r"it's", "it is", sentence)
    sentence = re.sub(r"that's", "that is", sentence)
    sentence = re.sub(r"what's", "that is", sentence)
    sentence = re.sub(r"where's", "where is", sentence)
    sentence = re.sub(r"how's", "how is", sentence)
    sentence = re.sub(r"\'ll", " will", sentence)
    sentence = re.sub(r"\'ve", " have", sentence)
    sentence = re.sub(r"\'re", " are", sentence)
    sentence = re.sub(r"\'d", " would", sentence)
    sentence = re.sub(r"\'re", " are", sentence)
    sentence = re.sub(r"won't", "will not", sentence)
    sentence = re.sub(r"can't", "cannot", sentence)
    sentence = re.sub(r"n't", " not", sentence)
    sentence = re.sub(r"n'", "ng", sentence)
    sentence = re.sub(r"'bout", "about", sentence)
    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")
    sentence = re.sub(r"[^a-zA-Z?.!,]+", " ", sentence)
    sentence = sentence.strip()
    return sentence
```

we haven't changed anything in this function which we will use later on. this function removes all the unnecessary spaces and punctuation that would interfere with the tokenization. The most important function is the load_conversation() which we have changed drastically as shown below.

```python
def load_conversations():
    speakers = []
    conversations = []
    inputs = []
    outputs = []
    id2line = {}
    for example in dataset["train"]:
        text = example["text"]
        # Split the text into lines
        lines = text.strip().split("\n")
        id2line[lines[0]] = lines[4]
        # Extract the speaker and conversation from each line
        for line in lines:
            # Find the index of the ":" delimiter
            delimiter_index = line.find(":")
            if delimiter_index != -1:
                speaker = line[:delimiter_index].strip()
                dialogue = line[delimiter_index + 1:].strip()
                speakers.append(speaker)
                conversations.append(dialogue)
        for i in range(0, len(conversations)):
           inputs.append(preprocess_sentence(conversations[:i]))
           outputs.append(preprocess_sentence(conversations[:i+1]))
    return inputs ,outputs

InD, OutData = load_conversations()
```

First we added four lists which are the speakers and conversations lists, inputs and outputs. these are empty lists that will be filled in the for loops made. the first for loop accessed the text key in the dictionary that holds the data. it splits the text into lines which will then have another for loop iterate over them. The for loop will iterate over each line and find the colon that separates the speaker and what they say (conversation) then If a delimiter is found, the speaker and dialogue are appended to the respective lists. Next, a loop runs from 0 to the length of `conversations`. Inside the loop, it creates inputs and outputs by calling the `preprocess_sentence` function on slices of the conversations. The `preprocess_sentence` function likely performs some text preprocessing or normalization. the loop then returns 'inputs' and 'outputs'.

The length of the input data was 9007 which we thought was reasonable. We then initialized the tokenizer and used the tokenize_and_filter function to tokenise our imput and output.

```
questions, answers = tokenize_and_filter(InD, OutData)
```

```
print(f"Vocab size: {VOCAB_SIZE}")
print(f"Number of samples: {len(questions)}")
```

```
Vocab size: 316
Number of samples: 13
```

as shown above, this has given us a vocab_size of 316 which we thought was reasonable as well.

Again we used the [tf.data.Dataset.from_tensor_slices]() function to put both inputs and outputs together and get a full dataset to be fed into the transfomer as shown below.

```
model.fit(dataset, epochs=EPOCHS)
```

The number of epochs was 250

```
model.fit(dataset, epochs=EPOCHS)
```

```
Epoch 1/250
1/1 [==============================] - 0s 40ms/step - loss: 0.8598 - accuracy: 0.2012
Epoch 2/250
1/1 [==============================] - 0s 37ms/step - loss: 0.8540 - accuracy: 0.1223
Epoch 3/250
1/1 [==============================] - 0s 40ms/step - loss: 0.8456 - accuracy: 0.1677
Epoch 4/250
1/1 [==============================] - 0s 38ms/step - loss: 0.8438 - accuracy: 0.1677
Epoch 5/250
1/1 [==============================] - 0s 41ms/step - loss: 0.8264 - accuracy: 0.2150
Epoch 6/250
1/1 [==============================] - 0s 37ms/step - loss: 0.8355 - accuracy: 0.1460
Epoch 7/250
1/1 [==============================] - 0s 37ms/step - loss: 0.8223 - accuracy: 0.1479
Epoch 8/250
```

```
Epoch 243/250
1/1 [==============================] - 0s 49ms/step - loss: 0.0990 - accuracy: 0.1696
Epoch 244/250
1/1 [==============================] - 0s 52ms/step - loss: 0.0983 - accuracy: 0.1677
Epoch 245/250
1/1 [==============================] - 0s 33ms/step - loss: 0.1071 - accuracy: 0.1677
Epoch 246/250
1/1 [==============================] - 0s 45ms/step - loss: 0.1109 - accuracy: 0.1677
Epoch 247/250
1/1 [==============================] - 0s 44ms/step - loss: 0.1015 - accuracy: 0.1657
Epoch 248/250
1/1 [==============================] - 0s 43ms/step - loss: 0.0924 - accuracy: 0.1677
Epoch 249/250
1/1 [==============================] - 0s 44ms/step - loss: 0.1021 - accuracy: 0.1657
Epoch 250/250
1/1 [==============================] - 0s 44ms/step - loss: 0.0983 - accuracy: 0.1696
<keras.callbacks.History at 0x7fc0b0f0c0a0>
```

the accuracy started off 0.2 which was alarming and ended up 0.16 which has led us to experiment a little by changing the hyperparameters onlt to fin that it wasn't learning much as well cause all experiments ended up with the following predictions

```
predict("where have you been!")
```

```
'a w a y , a w a y ! i f t h e y'
```

```
[ ] predict("It's a trap")
```

```
'a w a y , a w a y !'
```

```
[ ] predict("Before we proceed any further, hear me speak.")
```

```
# feed the model with its previous output
sentence = "I am not crazy, my mother had me tested."
for _ in range(5):
    print(f"Input: {sentence}")
    sentence = predict(sentence)
    print(f"Output: {sentence}\n")
```

```
Input: I am not crazy, my mother had me tested.
Output: a w a y , a w a y ! i f t h e a e l e l e a n n e s t h a t

Input: a w a y , a w a y ! i f t h e a e l e l e a n n e s t h a t
Output: a w a y , a w a y ! i f t h e y

Input: a w a y , a w a y ! i f t h e y
Output: a w a y , a w a y ! i f t h e y

Input: a w a y , a w a y ! i f t h e y
Output: a w a y , a w a y ! i f t h e y

Input: a w a y , a w a y ! i f t h e y
Output: a w a y , a w a y ! i f t h e y
```

this again was not satisfying and there was definitely something wrong so we decided to go with another approach which is approach 3.

Approach Three

approach three was us realizing that in the last two ways we have used a dataset with only 13 samples to train our models, that why they only outputed away every single time because away was repeated twice which led the model to think that away is most probable to show. so we changed the MAX_LENGHT which was at the beginning 40 and we increased it to 316.

Hyperparameters

To keep this example small and relatively fast, the values for *num_layers, d_model, and units* have been reduced. See the paper for all the other versions of the transformer.

```
[ ] # Maximum sentence length
    MAX_LENGTH = 316

    # Maximum number of samples to preprocess
    MAX_SAMPLES = 10500

    # For tf.data.Dataset
    BATCH_SIZE = 64 * strategy.num_replicas_in_sync
    BUFFER_SIZE = 2000

    # For Transformer
    NUM_LAYERS = 2 #(changed)
    D_MODEL = 128
    NUM_HEADS = 8
    UNITS = 256
    DROPOUT = 0.0001#(changed)

    EPOCHS = 250 #(changed)
```

BEFORE

```
print(f"Vocab size: {VOCAB_SIZE}")
print(f"Number of samples: {len(questions)}")
```

```
Vocab size: 316
Number of samples: 13
```

AFTER

```
print(f"Vocab size: {VOCAB_SIZE}")
print(f"Number of samples: {len(questions)}")
```

```
Vocab size: 316
Number of samples: 61
```

This has finally yielded more output

```
predict("Where have you been?")
```

```
'a w a y , a w a y ! i f t h e h e a l e a n e n e s t h e a t h e r t h e g h e g h e w o w a n o a o a g h e g h e g h e g t o m o n o n o n o n o h e a o f o f o f o n o n e n e n
e n e n e n e n e n e w e h e a w e a n a n a n a n a h e a e a t h e e e a l t h e a o  ! t h e a l e a l e a t h e t o t o r e a l e a w h e h a w a w o f t h e w e w e a t h e a t
h e a t h e a t h e h e h e a t h e a t h e h e a t h e a h e a a o u h e h e a o u o u o t h e a a h e a t h e t h e o u o u t h e n o t h e a n a n a a '
```

```
predict("It's a trap")
```

```
'a a ya ya ya yyyyt h e e e e e e e a e a h e s s h e a t h e m e h e g h e a '
```

```
predict("Let us kill him, and we'll have corn at our own price")
```

```
'a w a y , a w a y ! i f t h e t h e l e a n n n t h e t h e t h e r t h e g h e s o n o o o o o o o o o n o d o o t o t h e o t o t o n o n o t h e n o t h e n o f o t o n o n c e n c
e n c e n n n n n b e n b e h e a n o n o n a n n n h e n o t h e t h e t h e h e n o t h e l t h e l l t h e s t h e t o t h e s o u s o n o h a n o n o n o t h e n o f t h e n o t h
e s t h e n e n o t h e h e r e o t h e h e h e a t h e h e a t h e h e h e h e o u o t h e h e a t h e n o t h e t o u t h e n o t h e n o n o u a n o u a n o u o u o u o t h e n o u
o n o n o u t h e n o t h e s o u o u o u o u o u o u o u o u o u o u o u o u o u o u '
```

```
# feed the model with its previous output
sentence = "I am not crazy, my mother had me tested."
for _ in range(5):
    print(f"Input: {sentence}")
    sentence = predict(sentence)
    print(f"Output: {sentence}\n")
```

```
Input: I am not crazy, my mother had me tested.
Output: a w a y , a w a y ! yt h e w e l t h e a n e n e n s t h e r t h e r t h e s o d s o w e w e s o r a n o r y s o t h e n o t o t h e n o n o n o n o n o n o f o t h e n s n e r

Input: a w a y , a w a y ! yt h e w e l t h e a n e n e n s t h e r t h e r t h e s o d s o w e w e s o r a n o r y s o t h e n o t o t h e n o n o n o n o n o n o f o t h e n s n e n
Output: a w a y , a w a y ! i t h e w h e l t h e a n e s t h e s t h e r t h e g h e s o o o w o o o o a w e a y o o o o t o t h e o t o m o n o n o n o n o o o o o o o o o o o n o n c

Input: a w a y , a w a y ! i t h e w h e l t h e a n e s t h e s t h e r t h e g h e s o o o w o o o o a w e a y o o o o t o t h e o t o m o n o n o n o n o o o o o o o o o o o o n o n c e
```

but it didn't work since still the output begins with away away which has got us thinking that maybe it's the data which has led us to Approach four.
