Introduction

Text generation models have a long and complex history. In the early days of natural language processing, language models focused primarily on rule-based approaches that attempted to encode grammar and syntax into computer algorithms. However, these early models had limited success in generating natural-sounding language (Manning & Schütze, 1999).

Researchers then began to explore statistical approaches to language modelling, which involved training models on large corpora of text data and using statistical techniques to identify patterns and regularities in the data. One early example of this approach was the n-gram model, which used the frequencies of short sequences of words to predict the likelihood of a given word occurring in a particular context (Brown et al., 1992). However, n-gram model only predicts the probability of the next word in a sequence given the previous n-1 words. While n-gram models can capture some basic dependencies between adjacent words, they are unable to capture longer-range dependencies or more complex relationships between words that may be separated by several other words (Marino et al., 2006).

In the late 1990s, researchers began to explore neural network-based approaches to language modelling, which involved training models on large amounts of text data using artificial neural networks (e.g.,Word2Vec, Schuster & Paliwal, 1997). One of the earliest neural network-based language models was the Recurrent Neural Network (RNN, Schuster, 1999). RNNs are able to capture the sequential relationships between words in a sentence or document by using a "memory" component that allows the model to retain information about previous inputs. However, early RNN models suffered from the "vanishing gradient" problem, that is, the gradients used to update the model's weights during backpropagation become very small (Bengio et al., 1994). This can cause the model's weights to stop updating, preventing the model from effectively learning from the data. The vanishing gradient problem is particularly acute in RNNs because they have a "memory" component that allows them to retain information about previous inputs (Hochreiter, 2011). This can make it difficult for the model to learn long-term dependencies in the data, which are essential for language modelling.

To address the vanishing gradient problem, researchers have developed several techniques, including the Long Short-Term Memory (LSTM) network, which uses a more sophisticated memory component that is better able to preserve information over longer periods of time (Hochreiter & Schmidhuber, 1997). This is because the memory cell in an LSTM network can retain information over many time steps, allowing the network to capture more complex patterns in the data (Li & Qian, 2016). However, it also raises a potential problem with LSTMs: they can be more prone to overfitting than standard RNNs (Melis et al., 2019). This is because they have a larger number of parameters. This can be resolved by gated recurrent unit (GRU), which is similar to LSTM but has a simpler architecture. which can make it easier to train and less prone to overfitting (Chung et al., 2014). However, there are also some potential disadvantages to using GRU. One problem with GRU is that it may not be as powerful as LSTM in modelling long-term dependencies in the data (Feng et al., 2021). This is because GRU only has two gates, which may not be sufficient to capture more complex patterns in the data.

The GPT (Generative Pre-trained Transformer) family of models are based on the Transformer architecture, which was introduced as an alternative to RNNs for sequence modelling tasks (Vaswani et al., 2017). Unlike RNNs, which process input sequentially and suffer from the vanishing gradient problem, Transformers can process entire input sequences in parallel, making them more efficient for long sequences. One of the main advantages of transformer-based models for text generation is their ability to capture long-range dependencies and contextual information in the input sequence (Cho et al., 2014). This allows the models to generate text that is coherent and grammatically correct (Alec et al., 2018). This is because they can capture long-range dependencies and contextual information in the input sequence, allowing them to generate text that is coherent and grammatically correct (Brown et al., 2020).

For the task in hand we will use a miniature GPT-2 model for word-level text generation and a two layer LSTM to for character-level autocompletion. This is because word-level text generation requires long-range dependencies and contextual information in order to formulate the correct response (Brown et al., 2020). Character level text generation on the other hand, contains little contextual information that can be used for predictions. Hence we use a LSTM model to prevent overfitting while retain information over many time steps, allowing the network to capture more complex patterns in character sequences (Melis et al., 2019).

Task 1: Word-level text generation

Dependencies

```
# set up environments
! pip install keras_nlp
import os
import numpy as np
import re
import keras_nlp
import tensorflow as tf
from tensorflow import keras
# conneting to google drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting keras_nlp
  Downloading keras_nlp-0.4.1-py3-none-any.whl (466 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 466.8/466.8 kB 6.7 MB/s eta 0:00:00
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-packages (from keras_nlp) (1.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from keras_nlp) (1.22.4)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from keras_nlp) (23.1)
Collecting tensorflow-text (from keras_nlp)
  Downloading tensorflow_text-2.12.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (6.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.0/6.0 MB 67.9 MB/s eta 0:00:00
Requirement already satisfied: tensorflow-hub>=0.8.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow-text->keras_nlp) (0.13.0)
Requirement already satisfied: tensorflow<2.13,>=2.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow-text->keras_nlp) (2.12.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (1.6.3)
Requirement already satisfied: flatbuffers>=2.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (23.3.3)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (0.4.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (0.2.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (1.54.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (3.8.0)
Requirement already satisfied: jax>=0.3.15 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (0.4.8)
Requirement already satisfied: keras<2.13,>=2.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (2.12.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (16.0.0)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (3.3.0)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (67.7.2)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (1.16.0)
Requirement already satisfied: tensorboard<2.13,>=2.12 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (2.12.2)
Requirement already satisfied: tensorflow-estimator<2.13,>=2.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (2.12.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (2.3.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (4.5.0)
Requirement already satisfied: wrapt<1.15,>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (1.14.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (0.32.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.10/dist-packages (from astunparse>=1.6.0->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (0.40.0)
Requirement already satisfied: ml-dtypes>=0.0.3 in /usr/local/lib/python3.10/dist-packages (from jax>=0.3.15->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (0.1.0)
Requirement already satisfied: scipy>=1.7 in /usr/local/lib/python3.10/dist-packages (from jax>=0.3.15->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (1.10.1)
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp)
Requirement already satisfied: google-auth-oauthlib<1.1,>=0.5 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text->ke
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (3.4.3
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (2
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-t
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text->ker
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text->keras_nlp) (2.3.0
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tens
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tenso
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-te
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from google-auth-oauthlib<1.1,>=0.5->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorf
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->ten
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text-
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.10/dist-packages (from werkzeug>=1.0.1->tensorboard<2.13,>=2.12->tensorflow<2.13,>=2.12.0->tensorflow-text
Requirement already satisfied: pyasn1<0.6.0,>=0.4.6 in /usr/local/lib/python3.10/dist-packages (from pyasn1-modules>=0.2.1->google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflo
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib<1.1,>=0.5->tensorboard<2.13,>=2.12->te
Installing collected packages: tensorflow-text, keras_nlp
Successfully installed keras_nlp-0.4.1 tensorflow-text-2.12.1
Mounted at /content/drive
```

Hyperparameters

```python
# hyperparameters
# Data
BATCH_SIZE = 64
SEQ_LEN = 128
MIN_TRAINING_SEQ_LEN = 450

# Model
EMBED_DIM = 256
FEED_FORWARD_DIM = 256
NUM_HEADS = 3
NUM_LAYERS = 2
VOCAB_SIZE = 5000  # Limits parameters in model.

# Optmizer
learning_rate = 0.001
beta1 = 0.9
beta2 = 0.98
epsilon = 1e-9
weight_decay = 0.01
clip_norm = 1.0

# Training
EPOCHS = 30

# Inference
NUM_TOKENS_TO_GENERATE = 80
```

Data sets

We the SimpleBooks dataset which included 1,573 Gutenberg books (Huyen, 2019). This dataset is selected because it has one the smallest vocabulary size to word-level tokens ratio (about 98k) which is only 1/3 of the WikiText-103's, but having roughly the same amount of tokens (~100M). These properties made this dataset a ideal to fit to a small model.

```python
# data directories
keras.utils.get_file(
    origin="https://dldata-public.s3.us-east-2.amazonaws.com/simplebooks.zip",
    extract=True,
)
dir = os.path.expanduser("~/.keras/datasets/simplebooks/")

# Load simplebooks-92 train set and filter out short lines.
raw_train_ds = (
    tf.data.TextLineDataset(dir + "simplebooks-92-raw/train.txt")
    .filter(lambda x: tf.strings.length(x) > MIN_TRAINING_SEQ_LEN)
    .batch(BATCH_SIZE)
    .shuffle(buffer_size=256)
)

# Load simplebooks-92 validation set and filter out short lines.
raw_val_ds = (
    tf.data.TextLineDataset(dir + "simplebooks-92-raw/valid.txt")
    .filter(lambda x: tf.strings.length(x) > MIN_TRAINING_SEQ_LEN)
    .batch(BATCH_SIZE)
)

# Load simplebooks-2 validation set for test set and filter out short lines.
raw_test_ds = (
    tf.data.TextLineDataset(dir + "simplebooks-2-raw/valid.txt")
    .filter(lambda x: tf.strings.length(x) > MIN_TRAINING_SEQ_LEN)
    .batch(BATCH_SIZE)
)
```

```
Downloading data from https://dldata-public.s3.us-east-2.amazonaws.com/simplebooks.zip
282386239/282386239 [==============================] - 20s 0us/step
```

Tokenizer

We train the tokenizer from our data set with a give size of VOCAB_SIZE (5000) so that we can limit the vocab size as much as possible so that we can control the number of model parameters. We also need to minimise the number of out-of-vocabulary (OOV) items also don not want to include too few vocabulary.

```python
# Train tokenizer vocabulary

#"[PAD]" for padding sequences to SEQ_LEN. This token has index 0 in both reserved_tokens and vocab,
#since WordPieceTokenizer (and other layers) consider 0/vocab[0] as the default padding.
#"[UNK]" for OOV sub-words, which should match the default oov_token="[UNK]" in WordPieceTokenizer.
#"[BOS]" stands for beginning of sentence, but here technically it is a token representing the beginning of each line of training data.
vocab = keras_nlp.tokenizers.compute_word_piece_vocabulary(
    raw_train_ds,
    vocabulary_size=VOCAB_SIZE,
    lowercase=True,
    reserved_tokens=["[PAD]", "[UNK]", "[BOS]"],
)

tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary=vocab,
    sequence_length=SEQ_LEN,
    lowercase=True,
)
```

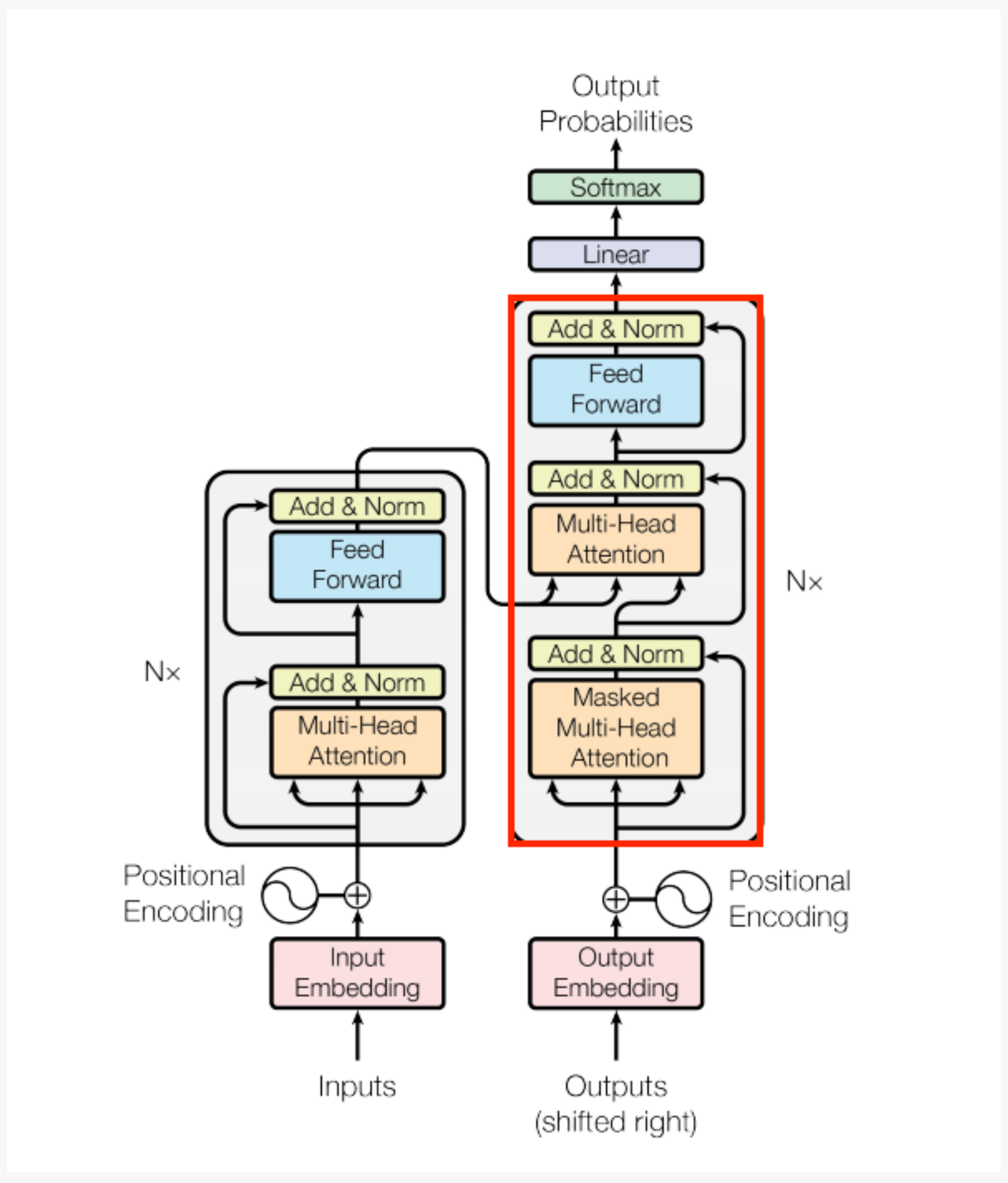
Tokenize Data

```
# pre-process the dataset by tokenizing and splitting it into features and labels.
# packer adds a start token
start_packer = keras_nlp.layers.StartEndPacker(
    sequence_length=SEQ_LEN,
    start_value=tokenizer.token_to_id("[BOS]"),
)


def preprocess(inputs):
    outputs = tokenizer(inputs)
    features = start_packer(outputs)
    labels = outputs
    return features, labels


# Tokenize and split into train and label sequences.
train_ds = raw_train_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE).prefetch(
    tf.data.AUTOTUNE
)
val_ds = raw_val_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE).prefetch(
    tf.data.AUTOTUNE
)
test_ds = raw_test_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE).prefetch(
    tf.data.AUTOTUNE
)
```

Build a miniature of a GPT-2 model

We created a down-scaled GPT model based on the architecture of the GPT-2 (Brown et al., 2020). We select a transformer-based model as the task is to generate two word after a sequence of words. Using a transformer-based model can process entire input sequences in parallel so that it can capture the long-range dependencies and contextual information in the input sequence. As we have limited computation resources we used a decoder-only transformer like the GPT-2 or Transformer XL (Dai et al., 2019). This is to save training time with little loss of performance. We use two-layer transformer decoder without cross attention (like GPT-2) as the task only require generating two words and the dataset we used contains relatively simple texts. A decoder-only model could be better at preventing overfitting. We use a two-layer decoder so that it is more robust to noise compare to a single layer of transformer decoder. The final model contains one embedding layer which combines the embedding for the token and its position. Two transformer decoder layer, with default casual masking so that the layers run with cross attention as it runs with decoder sequence only. One dense linear layer for outputs.

The decoder layer



```
inputs = keras.layers.Input(shape=(None,), dtype=tf.int32)

# Embedding.
embedding_layer = keras_nlp.layers.TokenAndPositionEmbedding(
    vocabulary_size=VOCAB_SIZE,
    sequence_length=SEQ_LEN,
    embedding_dim=EMBED_DIM,
    mask_zero=True,
)
x = embedding_layer(inputs)

# Transformer decoders.
for _ in range(NUM_LAYERS):
    decoder_layer = keras_nlp.layers.TransformerDecoder(
        num_heads=NUM_HEADS,
        intermediate_dim=FEED_FORWARD_DIM,
    )
    x = decoder_layer(x)  # Giving one argument only skips cross-attention.

# Output.
outputs = keras.layers.Dense(VOCAB_SIZE)(x)
model = keras.Model(inputs=inputs, outputs=outputs)
optimizer = tf.keras.optimizers.Adam(
    learning_rate=learning_rate,
    beta_1=beta1,
    beta_2=beta2,
    epsilon=epsilon,
    weight_decay=weight_decay,
    clipnorm=clip_norm
)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
perplexity = keras_nlp.metrics.Perplexity(from_logits=True, mask_token_id=0)
model.compile(optimizer=optimizer, loss=loss_fn, metrics=[perplexity])
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, None)]            0

 token_and_position_embeddin  (None, None, 256)        1312768
 g (TokenAndPositionEmbeddin
 g)

 transformer_decoder (Transf  (None, None, 256)        394749
 ormerDecoder)

 transformer_decoder_1 (Tran  (None, None, 256)        394749
 sformerDecoder)

 dense (Dense)               (None, None, 5000)        1285000

=================================================================
Total params: 3,387,266
Trainable params: 3,387,266
Non-trainable params: 0
_________________________________________________________________
```

Train the Model

```
training_info1 = model.fit(train_ds, validation_data=val_ds , epochs=EPOCHS)
```

```
Epoch 1/30
3169/3169 [==============================] - 373s 111ms/step - loss: 4.6045 - perplexity: 100.3283 - val_loss: 4.2143 - val_perplexity: 68.3144
Epoch 2/30
3169/3169 [==============================] - 238s 73ms/step - loss: 4.0951 - perplexity: 60.2832 - val_loss: 4.0396 - val_perplexity: 57.3481
Epoch 3/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.9813 - perplexity: 53.7913 - val_loss: 3.9608 - val_perplexity: 52.9081
Epoch 4/30
3169/3169 [==============================] - 238s 73ms/step - loss: 3.9172 - perplexity: 50.4474 - val_loss: 3.9331 - val_perplexity: 51.5475
Epoch 5/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.8746 - perplexity: 48.3431 - val_loss: 3.8792 - val_perplexity: 48.7406
Epoch 6/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.8413 - perplexity: 46.7591 - val_loss: 3.8640 - val_perplexity: 47.9517
Epoch 7/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.8170 - perplexity: 45.6355 - val_loss: 3.8495 - val_perplexity: 47.3524
Epoch 8/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7961 - perplexity: 44.6891 - val_loss: 3.8346 - val_perplexity: 46.5903
Epoch 9/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7779 - perplexity: 43.8857 - val_loss: 3.8193 - val_perplexity: 45.9149
Epoch 10/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7632 - perplexity: 43.2430 - val_loss: 3.8023 - val_perplexity: 45.0598
Epoch 11/30
3169/3169 [==============================] - 240s 73ms/step - loss: 3.7502 - perplexity: 42.6870 - val_loss: 3.7826 - val_perplexity: 44.2496
Epoch 12/30
3169/3169 [==============================] - 240s 73ms/step - loss: 3.7387 - perplexity: 42.1982 - val_loss: 3.7684 - val_perplexity: 43.6586
Epoch 13/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7284 - perplexity: 41.7631 - val_loss: 3.7364 - val_perplexity: 42.2854
Epoch 14/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7207 - perplexity: 41.4410 - val_loss: 3.7795 - val_perplexity: 44.0846
Epoch 15/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7120 - perplexity: 41.0827 - val_loss: 3.7736 - val_perplexity: 43.8862
Epoch 16/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.7051 - perplexity: 40.8029 - val_loss: 3.7653 - val_perplexity: 43.5133
Epoch 17/30
3169/3169 [==============================] - 240s 74ms/step - loss: 3.6998 - perplexity: 40.5874 - val_loss: 3.7487 - val_perplexity: 42.8403
Epoch 18/30
3169/3169 [==============================] - 240s 73ms/step - loss: 3.6946 - perplexity: 40.3766 - val_loss: 3.7338 - val_perplexity: 42.1590
Epoch 19/30
3169/3169 [==============================] - 240s 73ms/step - loss: 3.6898 - perplexity: 40.1834 - val_loss: 3.7489 - val_perplexity: 42.8151
Epoch 20/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.6857 - perplexity: 40.0185 - val_loss: 3.7216 - val_perplexity: 41.6486
Epoch 21/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.6813 - perplexity: 39.8417 - val_loss: 3.7264 - val_perplexity: 41.8366
Epoch 22/30
3169/3169 [==============================] - 240s 73ms/step - loss: 3.6778 - perplexity: 39.7041 - val_loss: 3.7360 - val_perplexity: 42.2831
Epoch 23/30
3169/3169 [==============================] - 239s 73ms/step - loss: 3.6751 - perplexity: 39.5964 - val_loss: 3.7364 - val_perplexity: 42.2689
Epoch 24/30
3169/3169 [==============================] - 238s 73ms/step - loss: 3.6723 - perplexity: 39.4839 - val_loss: 3.7286 - val_perplexity: 42.0576
Epoch 25/30
3169/3169 [==============================] - 237s 73ms/step - loss: 3.6699 - perplexity: 39.3879 - val_loss: 3.7067 - val_perplexity: 41.0853
Epoch 26/30
3169/3169 [==============================] - 237s 73ms/step - loss: 3.6670 - perplexity: 39.2748 - val_loss: 3.7271 - val_perplexity: 41.8819
Epoch 27/30
3169/3169 [==============================] - 238s 73ms/step - loss: 3.6646 - perplexity: 39.1797 - val_loss: 3.7162 - val_perplexity: 41.4692
Epoch 28/30
3169/3169 [==============================] - 238s 73ms/step - loss: 3.6623 - perplexity: 39.0902 - val_loss: 3.7134 - val_perplexity: 41.3435
Epoch 29/30
3169/3169 [==============================] - 237s 73ms/step - loss: 3.6596 - perplexity: 38.9876 - val_loss: 3.7225 - val_perplexity: 41.6976
```

Test the Model

We use perplexity (the lower the better) as the metric to evaluate the efficacy of our model. It represents probability for a sentence to be produced by the model trained on a dataset. The result indicates that our model successfully learned the features in the dataset and the learned model can be fit to other datasets with neglectable losses.

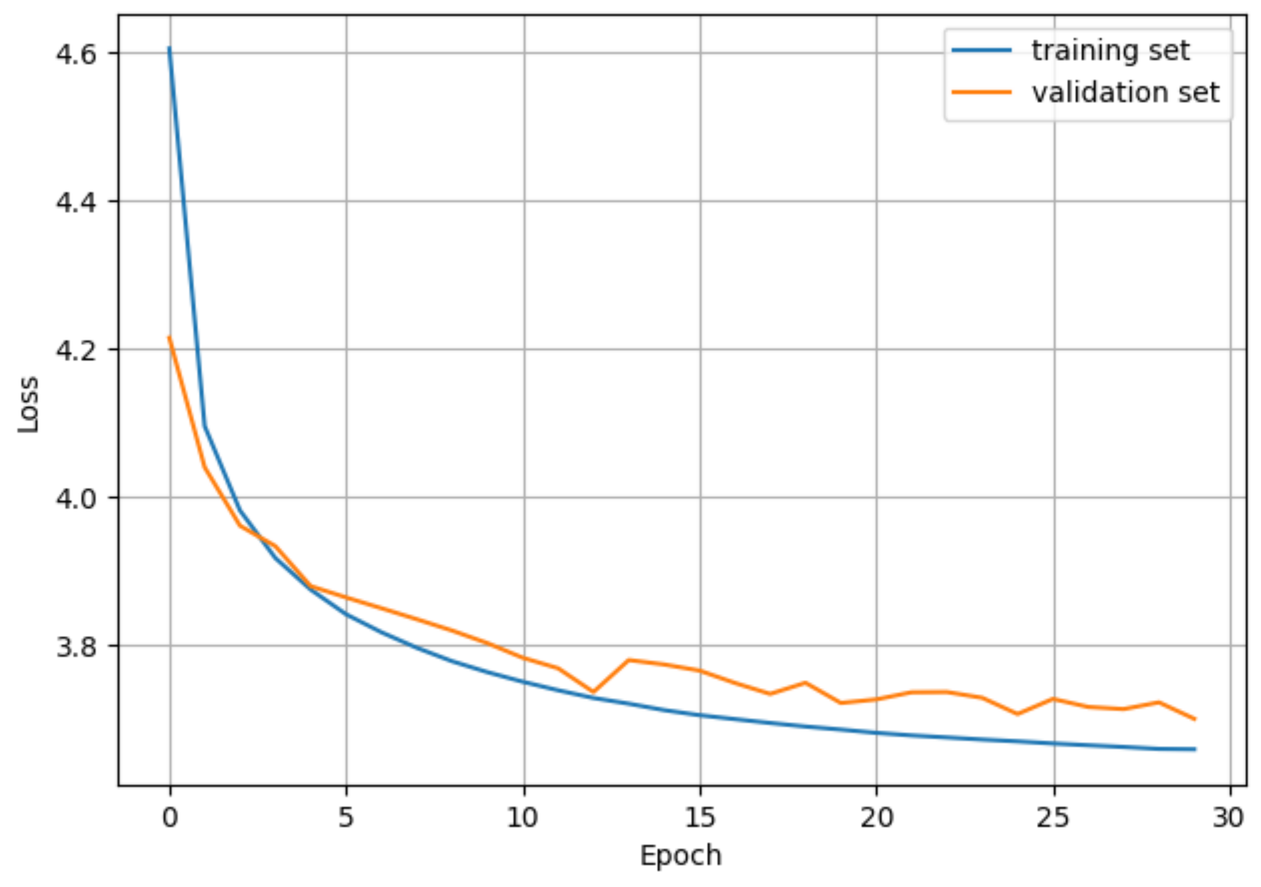
```
# For text generative models if the model is completely dumb (the worst result) then the perplexity should be the size of the vocabulary (in our model 5000)
# test
test_loss = model.evaluate(test_ds, verbose=1)
print("Test perplexity:", test_loss)
```

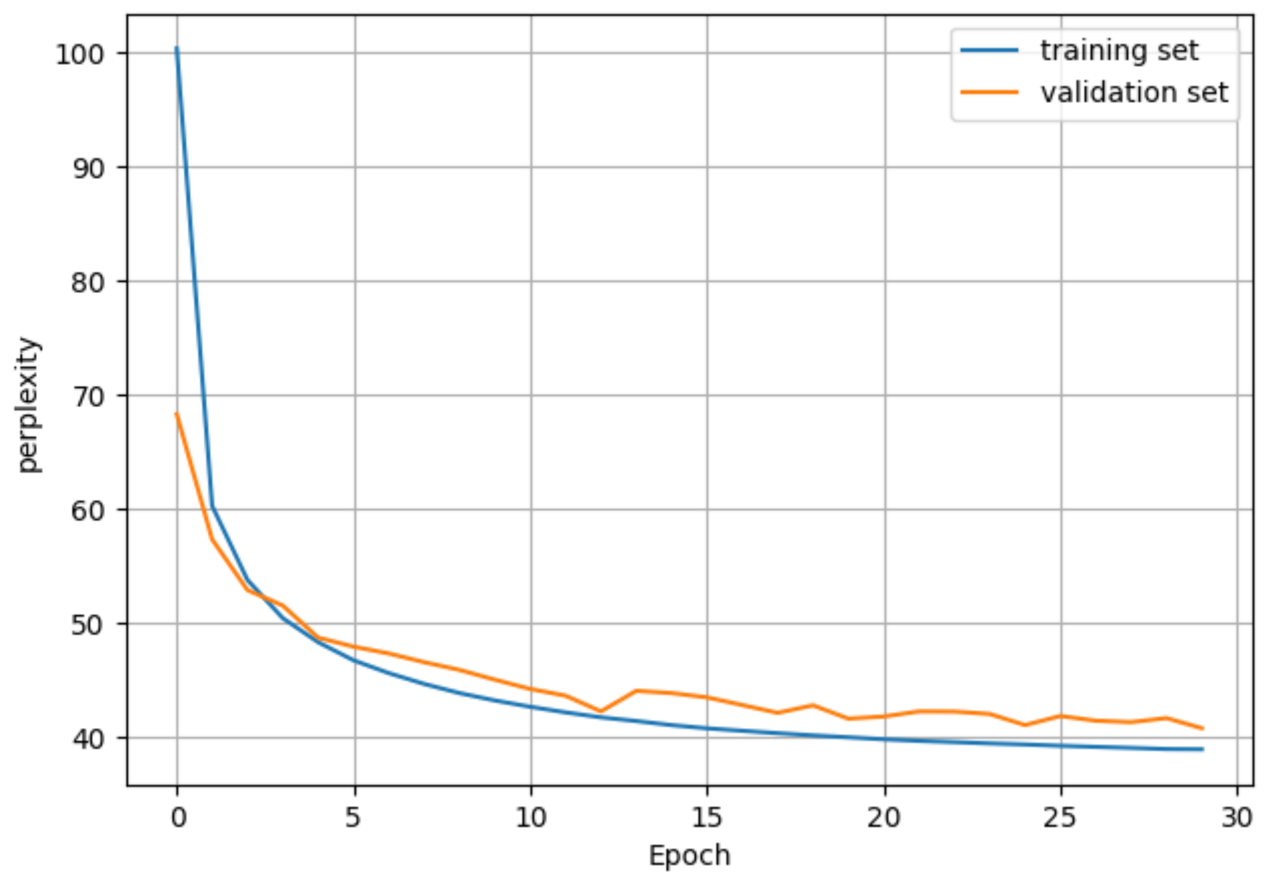
```
1/1 [==============================] - 1s 730ms/step - loss: 3.7002 - perplexity: 40.8141
Test perplexity: [3.7002315521240234, 40.814117431640625]
```

Visualise Training History

The plotted training history indicates a significant decrease in perplexity over 30 epochs this means that the language model has become more accurate at predicting the test set over the course of training. The overfitting is at a acceptable level.

```
# visualise the training graphs, compare the training and evaluation errors
import matplotlib.pyplot as plt
%matplotlib inline
def plot_training_history(training_info):
  fig, axs = plt.subplots(1, 2, figsize=(16, 5))
  axs[0].plot(training_info1.history['loss'], label="training set")
  axs[0].plot(training_info1.history['val_loss'], label="validation set")
  axs[0].set_xlabel("Epoch")
  axs[0].set_ylabel("Loss")
  axs[0].grid(True)
  axs[0].legend()
  try:
    axs[1].plot(training_info1.history['perplexity'], label="training set")
    axs[1].plot(training_info1.history['val_perplexity'], label="validation set")
    axs[1].set_xlabel("Epoch")
    axs[1].set_ylabel("perplexity")
    axs[1].grid(True)
    axs[1].legend()
  except:
    pass
  plt.show()

plot_training_history(training_info1)
```





```
# save the model

if not os.path.exists('/content/drive/MyDrive/saved_models'):
    os.makedirs('/content/drive/MyDrive/saved_models')
model.save('/content/drive/MyDrive/saved_models/task1_model')
task1_model = keras.models.load_model('/content/drive/MyDrive/saved_models/task1_model')
```

```
WARNING:absl:Found untraced functions such as token_embedding1_layer_call_fn, token_embedding1_layer_call_and_return_conditional_losses, position_embedding1_layer_call_fn, position_
```

Test With an Example

```
# genearted a text with [BOS] token at the padded at the start
inputs = "sun was shining, and the sky was"

prompt_tokens = tf.convert_to_tensor(np.trim_zeros(preprocess(inputs)[0].numpy()))
```

```
# packer adds a start token
start_packer = keras_nlp.layers.StartEndPacker(
    sequence_length=SEQ_LEN,
    start_value=tokenizer.token_to_id("[BOS]"),
)


def preprocess(inputs):
    outputs = tokenizer(inputs)
    features = start_packer(outputs)
    labels = outputs
    return features, labels


# Tokenize and split into train and label sequences.
train_ds = raw_train_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE).prefetch(
    tf.data.AUTOTUNE
)
val_ds = raw_val_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE).prefetch(
    tf.data.AUTOTUNE
)
```

```
def token_logits_fn(inputs):

    cur_len = inputs.shape[1]
    output = model(inputs)
    return output[:, cur_len - 1, :]  # return next token logits
```

```
# beam search geneartion
output_tokens = keras_nlp.utils.beam_search(
    token_logits_fn,
    prompt_tokens,
    max_length=NUM_TOKENS_TO_GENERATE,
    num_beams=1,
    from_logits=True,
)
txt = tokenizer.detokenize(output_tokens)
output = str(txt)[18:].split(" ")[0:len(inputs.split(" "))+3]
print(' '.join(output))
#print(f"Beam search generated text: \n{txt}\n")
```

```
sun was shining , and the sky was cloudless ,
```

with out put : "blue" "and", this model fullfilled word-level generative task.

Task 2: Character level Generation

Dependencies

```
import io
import re
import random
import string
import numpy as np
import nltk
import keras
import tensorflow as tf
import datetime
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.layers import Dense, LSTM, Dropout, Embedding
from tensorflow.keras.callbacks import EarlyStopping
from nltk.lm import Vocabulary
from nltk.corpus import gutenberg
from tensorboard.plugins.hparams import api as hp
def get_device():
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')
```

Process the text data

We used the first four books from the Gutenberg corpus in the NLTK corpora as character level generative model requires less data to train. We keep only English characters and the space mark and convert all letters to lower cases.

```
# Download the Gutenberg dataset from NLTK
nltk.download('gutenberg')
# Load the Gutenberg dataset
files = gutenberg.fileids()
text = nltk.corpus.gutenberg.raw(files[0:5])
text_length = len(text)
text = re.sub(r'[^a-zA-Z ]+', '', text)
# Remove \n
text = text.replace('\n', '')
# Convert to lowercase
text = text.lower()
print("Corpus length:", len(text))
# calculate the index for the 20th percentile for test set
chars = sorted(set(text))
print("Total chars:", len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
total_index = [char_indices[char] for char in text]
total_index[:10]
```

```
[nltk_data] Downloading package gutenberg to /root/nltk_data...
[nltk_data]   Unzipping corpora/gutenberg.zip.
Corpus length: 5916172
Total chars: 27
[5, 13, 13, 1, 0, 2, 25, 0, 10, 1]
```

Split to train and test data sets

```
train_index = total_index[10000:]
test_index = total_index[:10000]
# test the the tokenizer
''.join(indices_char[i] for i in total_index[:10])
```

```
'emma by ja'
```

Create inputs and outputs

As we are predicting characters, we need to create an array of the first few (e.g., 10) characters each acting as an input and the last character as the output. For example, for the text 'this and that' The input will be -> [['t', ' '], ['h', 't'], ['i', 'h'], ['s', 'a'], [' ', 't'], ['a'], ['n']] -> but instead of the characters, there will be the index of the character. And the output will be -> ['d']

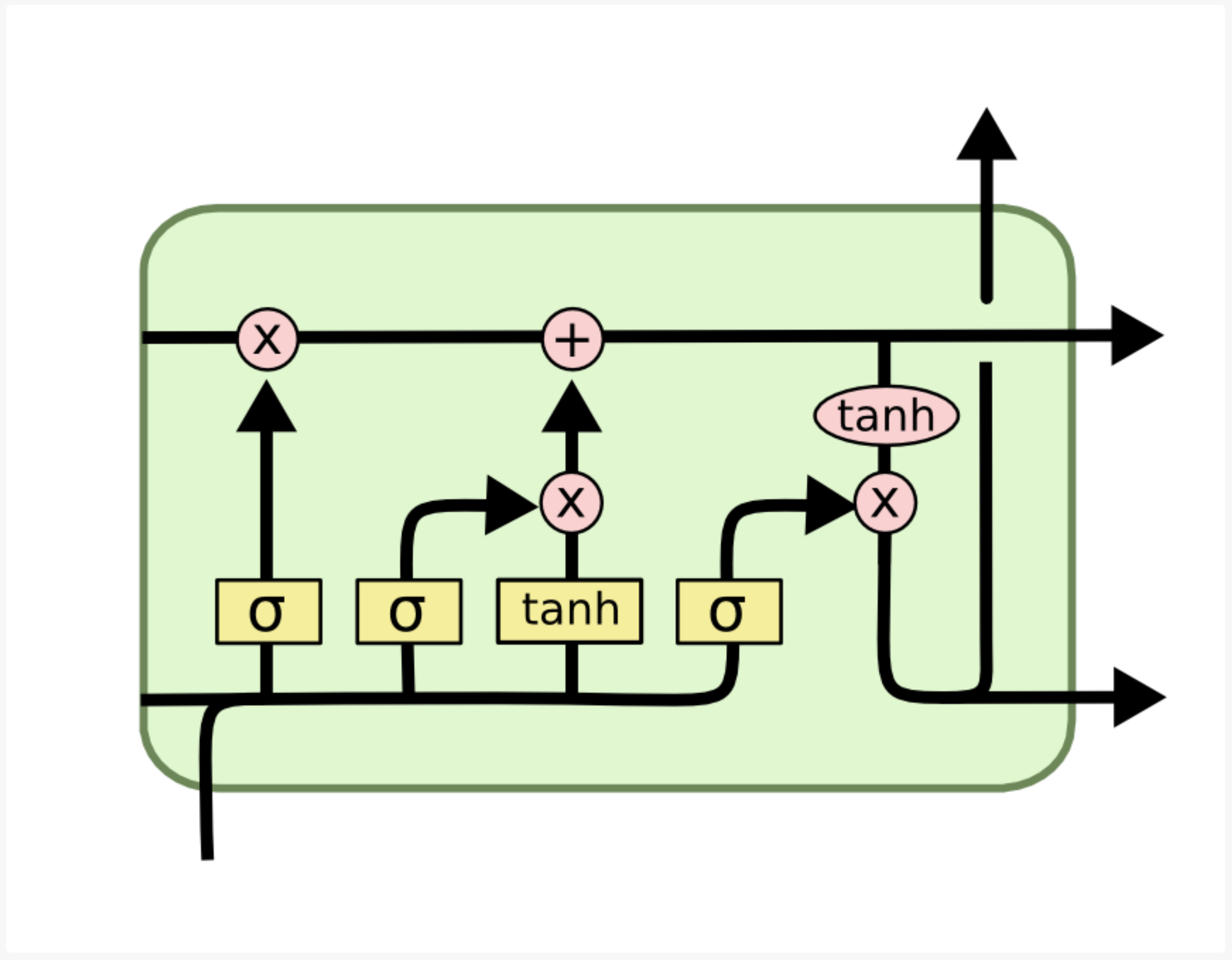
```
pred_num = 10
xin_train = [[train_index[j+i] for j in range(0, len(train_index)-1-pred_num, pred_num)] for i in range(pred_num)]
y_train = [train_index[i+pred_num] for i in range(0, len(train_index)-1-pred_num, pred_num)]
xin_test = [[test_index[j+i] for j in range(0, len(test_index)-1-pred_num, pred_num)] for i in range(pred_num)]
y_test = [test_index[i+pred_num] for i in range(0, len(test_index)-1-pred_num, pred_num)]
X_train = [np.stack(xin_train[i][:-2]) for i in range(pred_num)]
Y_train = np.stack(y_train[:-2])
X_test = [np.stack(xin_test[i][:-2]) for i in range(pred_num)]
Y_test = np.stack(y_test[:-2])
X_train[1].shape, Y_train.shape
```

```
((590615,), (590615,))
```

Build a two layer LSTM

We build a LSTM for character level text generation task. To predict the next two character in any give sequence (e.g., “rry christm_” -> as). Compare to convectional RNN models LSTM is more robust to vanishing gradient problems. Moreover, we segmented inputs to the length of ten hence, this further prevents vanishing gradient. However, consider our relatively small data set is more likely to overfit during training, we added drop off layers that randomly drops (sets to zero) a fraction of the output features of a layer during training. This helps to prevent the model from overfitting by introducing noise into the output of each layer and forcing the model to learn more robust representations that are less sensitive to small variations in the input. The final model consists of a embedding layer which combines the embedding for the length of the input, the token, and its position. 2 LSTM layers and two drop off layers. And a dense linear layer for outputs

The LSTM



```
hidden_layers = 256
vocab_size = 27
n_fac = 42
model = Sequential([
        Embedding(vocab_size, n_fac, input_length=pred_num),
        layers.LSTM(128, return_sequences = True, activation='tanh'),
        layers.Dropout(.2),
        layers.LSTM(128, return_sequences = False, activation='tanh'),
        layers.Dropout(.2),
        Dense(2*vocab_size, activation='softmax')
    ])
optimizer = keras.optimizers.RMSprop(learning_rate=0.002)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=['accuracy'])
```

```
model.build()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding (Embedding)       (None, 10, 42)            1134      
                                                                 
 lstm (LSTM)                 (None, 10, 128)           87552     
                                                                 
 dropout_6 (Dropout)         (None, 10, 128)           0         
                                                                 
 lstm_1 (LSTM)               (None, 128)               131584    
                                                                 
 dropout_7 (Dropout)         (None, 128)               0         
                                                                 
 dense_5 (Dense)             (None, 54)                6966      
                                                                 
=================================================================
Total params: 227,236
Trainable params: 227,236
Non-trainable params: 0
_________________________________________________________________
```

Train the model

```
epochs = 40
batch_size = 256

training_info = model.fit(np.stack(X_train, 1), Y_train, batch_size=batch_size, epochs = epochs, validation_split=0.10, callbacks=EarlyStopping(monitor="val_loss", patience=15))
print()
```

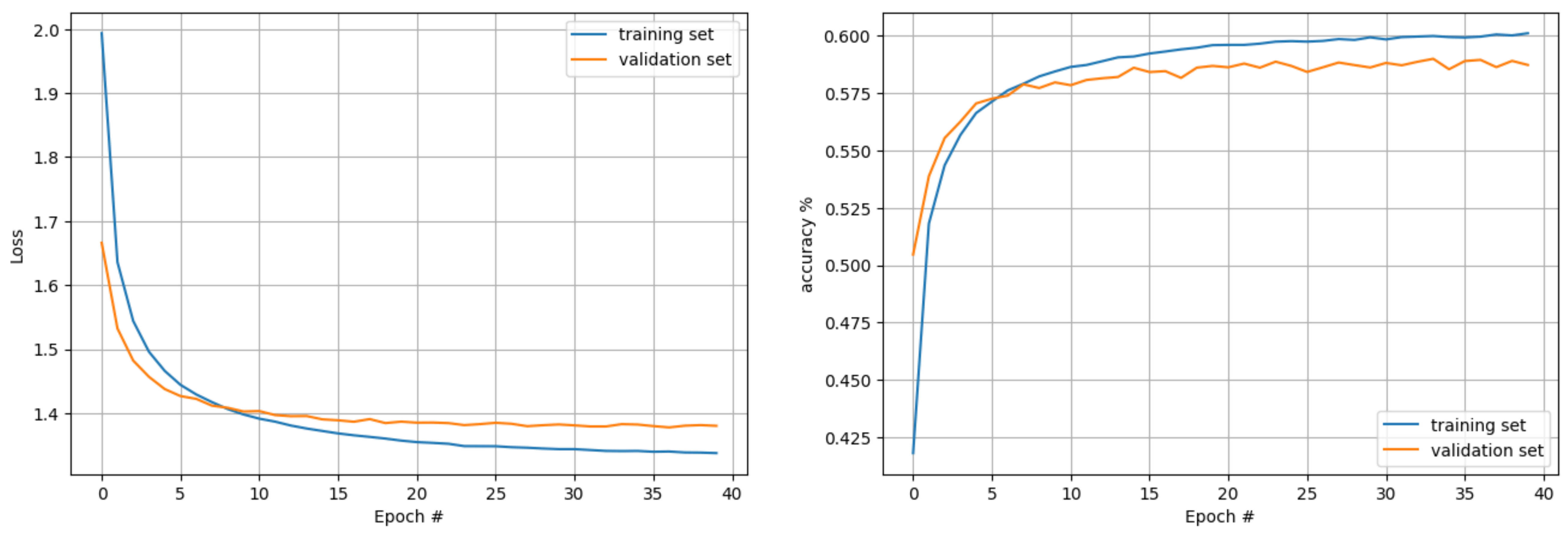
```
Epoch 1/40
2077/2077 [==============================] - 18s 7ms/step - loss: 1.9935 - accuracy: 0.4182 - val_loss: 1.6662 - val_accuracy: 0.5047
Epoch 2/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.6360 - accuracy: 0.5182 - val_loss: 1.5323 - val_accuracy: 0.5388
Epoch 3/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.5440 - accuracy: 0.5436 - val_loss: 1.4821 - val_accuracy: 0.5555
Epoch 4/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.4959 - accuracy: 0.5568 - val_loss: 1.4568 - val_accuracy: 0.5626
Epoch 5/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.4660 - accuracy: 0.5664 - val_loss: 1.4377 - val_accuracy: 0.5706
Epoch 6/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.4442 - accuracy: 0.5713 - val_loss: 1.4267 - val_accuracy: 0.5726
Epoch 7/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.4292 - accuracy: 0.5762 - val_loss: 1.4224 - val_accuracy: 0.5740
Epoch 8/40
```

```
2077/2077 [==============================] - 13s 6ms/step - loss: 1.4172 - accuracy: 0.5791 - val_loss: 1.4117 - val_accuracy: 0.5789
Epoch 9/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.4064 - accuracy: 0.5823 - val_loss: 1.4084 - val_accuracy: 0.5773
Epoch 10/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3980 - accuracy: 0.5845 - val_loss: 1.4026 - val_accuracy: 0.5797
Epoch 11/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3914 - accuracy: 0.5865 - val_loss: 1.4033 - val_accuracy: 0.5785
Epoch 12/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3868 - accuracy: 0.5873 - val_loss: 1.3970 - val_accuracy: 0.5808
Epoch 13/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3807 - accuracy: 0.5889 - val_loss: 1.3953 - val_accuracy: 0.5815
Epoch 14/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3761 - accuracy: 0.5906 - val_loss: 1.3955 - val_accuracy: 0.5821
Epoch 15/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3722 - accuracy: 0.5910 - val_loss: 1.3904 - val_accuracy: 0.5861
Epoch 16/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3684 - accuracy: 0.5923 - val_loss: 1.3889 - val_accuracy: 0.5842
Epoch 17/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3654 - accuracy: 0.5932 - val_loss: 1.3866 - val_accuracy: 0.5846
Epoch 18/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3630 - accuracy: 0.5941 - val_loss: 1.3908 - val_accuracy: 0.5817
Epoch 19/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3603 - accuracy: 0.5948 - val_loss: 1.3845 - val_accuracy: 0.5861
Epoch 20/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3572 - accuracy: 0.5959 - val_loss: 1.3868 - val_accuracy: 0.5869
Epoch 21/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3548 - accuracy: 0.5960 - val_loss: 1.3852 - val_accuracy: 0.5863
Epoch 22/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3536 - accuracy: 0.5960 - val_loss: 1.3853 - val_accuracy: 0.5879
Epoch 23/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3522 - accuracy: 0.5966 - val_loss: 1.3845 - val_accuracy: 0.5861
Epoch 24/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3485 - accuracy: 0.5975 - val_loss: 1.3814 - val_accuracy: 0.5887
Epoch 25/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3484 - accuracy: 0.5977 - val_loss: 1.3829 - val_accuracy: 0.5869
Epoch 26/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3483 - accuracy: 0.5975 - val_loss: 1.3850 - val_accuracy: 0.5843
Epoch 27/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3469 - accuracy: 0.5978 - val_loss: 1.3834 - val_accuracy: 0.5863
Epoch 28/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3460 - accuracy: 0.5986 - val_loss: 1.3797 - val_accuracy: 0.5884
Epoch 29/40
2077/2077 [==============================] - 13s 6ms/step - loss: 1.3448 - accuracy: 0.5982 - val_loss: 1.3812 - val_accuracy: 0.5872
```

Visualise Training History

The training history suggested that the model's accuracy improved significantly over 40 epochs with neglectable overfitting level.

```
import matplotlib.pyplot as plt
%matplotlib inline
def plot_training_history(training_info):
  fig, axs = plt.subplots(1, 2, figsize=(16, 5))
  axs[0].plot(training_info.history['loss'], label="training set")
  axs[0].plot(training_info.history['val_loss'], label="validation set")
  axs[0].set_xlabel("Epoch #")
  axs[0].set_ylabel("Loss")
  axs[0].grid(True)
  axs[0].legend()
  try:
    axs[1].plot(training_info.history['accuracy'], label="training set")
    axs[1].plot(training_info.history['val_accuracy'], label="validation set")
    axs[1].set_xlabel("Epoch #")
    axs[1].set_ylabel("accuracy %")
    axs[1].grid(True)
    axs[1].legend()
  except:
    pass
  plt.show()

plot_training_history(training_info)
```



Test with Example

```
def predict_next_char(inp):
    inp = inp[-pred_num:]
    index = [char_indices[i] for i in inp]
    arr = np.expand_dims(np.array(index), axis=0)
    prediction = model.predict(arr)
    first_char = indices_char[np.argmax(prediction[-pred_num:])]
    out1 = inp + first_char
    second_char = indices_char[np.argmax(out1[-pred_num:])]
    combi = first_char + second_char
    return combi, first_char, second_char, out1, inp + combi
```

```
predict_next_char('emma by ja')
```

```
1/1 [==============================] - 1s 619ms/step
('n ', 'n', ' ', 'emma by jan', 'emma by jan ')
```

```
predict_next_char('emma by ja')
```

```
1/1 [==============================] - 1s 590ms/step
('c ', 'c', ' ', 'emma by jac', 'emma by jac ')
```

```
predict_next_char('rry christm')
```

```
1/1 [==============================] - 0s 17ms/step
('a ', 'a', ' ', 'ry christma', 'ry christma ')
```

```
from google.colab import drive
drive.mount('/content/drive')
import os

if not os.path.exists('/content/drive/MyDrive/saved_models'):
    os.makedirs('/content/drive/MyDrive/saved_models')
model.save('/content/drive/MyDrive/saved_models/task2_model')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
WARNING:absl:Found untraced functions such as lstm_cell_layer_call_fn, lstm_cell_layer_call_and_return_conditional_losses, lstm_cell_1_layer_call_fn, lstm_cell_1_layer_call_and_retu
```

```
task2_model = keras.models.load_model('/content/drive/MyDrive/saved_models/task2_model')
```

Test the Model

The model's accuracy stays at the same level in the test set, with acceptable loss in accuracy.

```
#(b)
test_loss = model.evaluate(np.stack(X_test, 1), Y_test, verbose=1)
print("Test loss:", test_loss)
```

```
32/32 [==============================] - 0s 3ms/step - loss: 1.4512 - accuracy: 0.5717
Test loss: [1.4511544704437256, 0.5717151165008545]
```

Discussion

For the first task we used a decoder-only transformer and t a LSTM in the second task, which are both common models for sequence generation tasks, such as language modelling or text generation. There are several differences in between the models: decoder-only Transformer is consisted on stacks of transformer decoders and LSTM with stacks of LSTM layers. The Transformer uses self-attention mechanisms to capture the dependencies between words in a sentence, while LSTMs use recurrent connections. Self-attention allows the model to capture global dependencies between words in a sentence, whereas recurrent connections are better at capturing local dependencies. The Transformer can process the input sequence in parallel, while LSTMs process the input sequence sequentially. This makes the Transformer faster than LSTMs, especially for long sequences. The Transformer has a larger memory capacity than LSTMs, which makes it better suited for modelling long-term dependencies in the input sequence. Transformers are better suited for tasks that require capturing global dependencies between words in a sentence hence were used for the first task, while LSTMs are better suited for tasks that require capturing local dependencies (used in second task). Transformers are faster and more memory-efficient than LSTMs, but LSTMs may be better for small datasets where overfitting is a concern (e.g., dataset in task 2).

One potential way of improving the accuracy in task 2 is to use a longer input sequence like what we did in task one. This would allow the model to capture more complex features in the global input and make more accurate generations. using GRUs for task2 may also help in imporving accuracy. However, the nature of task2 determined that the input question itself is very noisy, with a sequence length of 11 there are many potentially correct predictions. Hence the accuracy for task 2 in bound to be relatively low when the input sequence is short.

Word-level generation involves predicting the probability distribution over a vocabulary of words given some input context, and then sampling from that distribution to generate the next word in the sequence. This approach models the text at the level of individual words, and can capture semantic relationships between words. However, it can also be more difficult to model OOV words, and may require a larger vocabulary size in our model we limited the vocabulary to 5000 to speed up training. On the other hand, character-level generation involves predicting the probability distribution over a vocabulary of characters given some input context, and then sampling from that distribution to generate the next character in the sequence. This approach models the text at the level of individual characters, and has the potential to capture more detailed information about the structure of the text, such as spelling and punctuation. However, our task was to predict the next two character in a very short (11 character include the space) sequence of text. Which requires ability to capture local level features over global features. Therefore, for the given task. Word level text generation is harder compare to character level generation as the contectual information in the input is more complex. Word-level generation is more commonly used for tasks such as language modelling, text classification, and machine translation, where capturing the meaning of individual words is important. Character-level generation is more commonly used for tasks such as handwriting recognition, text-to-speech synthesis, and text style transfer, where the structure of the text is important.

References:

Alec, R., Karthik, N., Tim, S., & Ilya, S. (2018). Improving Language Understanding by Generative Pre-Training. https://gluebenchmark.com/leaderboard

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning Long-Term Dependencies with Gradient Descent is Difficult. IEEE Transactions on Neural Networks, 5(2), 157–166. https://doi.org/10.1109/72.279181

Brown, P., DeSouza, P., Mercer, R., Pietra, V. J., & Lai, J. (1992). Class-based n-gram models of natural language. Computational Linguistics. https://doi.org/10.5555/176313.176316

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., … Amodei, D. (2020). Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems, 2020-December. https://arxiv.org/abs/2005.14165v4

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 1724–1734. https://doi.org/10.3115/v1/d14-1179

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. https://arxiv.org/abs/1412.3555v1

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., & Salakhutdinov, R. (2019). Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. ACL 2019 - 57th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, 2978–2988. https://doi.org/10.18653/v1/p19-1285

Feng, Y., Hu, B., Gong, Y., Sun, F., Liu, Q., & Ou, W. (2021). GRN: Generative Rerank Network for Context-wise Recommendation. ACM Reference Format. https://doi.org/10.1145/1122445.1122456

Hochreiter, S. (2011). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. Https://Doi.Org/10.1142/S0218488598000094, 6(2), 107–116. https://doi.org/10.1142/S0218488598000094

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735–1780. https://doi.org/10.1162/NECO.1997.9.8.1735

Huyen, H. (2019). SimpleBooks: Long-term dependency book dataset with simplified English vocabulary for word-level language modeling. https://arxiv.org/abs/1911.12391v1

Li, D., & Qian, J. (2016). Text sentiment analysis based on long short-term memory. 2016 1st IEEE International Conference on Computer Communication and the Internet, ICCCI 2016, 471–475. https://doi.org/10.1109/CCI.2016.7778967

Manning, C. D., & Schütze, H. (1999). Foundations of statistical natural language processing. In MIT Press (Vol. 1).

Marino, J. B., Banchs, R. E., Crego, J. M., De Gispert, A., Lambert, P., Fonollosa, J. A. R., & Costa-Jussà, M. R. (2006). N-gram-based Machine Translation. Computational Linguistics, 32(4), 527–549. https://doi.org/10.1162/COLI.2006.32.4.527

Melis, G., Kočiský, T., & Blunsom, P. (2019). Mogrifier LSTM. https://arxiv.org/abs/1909.01792v2

Mujika, A., Meier, F., & Steger, A. (2017). Fast-Slow Recurrent Neural Networks. Advances in Neural Information Processing Systems, 2017-December, 5916–5925. https://arxiv.org/abs/1705.08639v2

Schuster, M. (1999). Better Generative Models for Sequential Data Problems: Bidirectional Recurrent Mixture Density Networks. Advances in Neural Information Processing Systems, 12.

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673–2681. https://doi.org/10.1109/78.650093

Vaswani, A., Brain, G., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All you Need. Advances in Neural Information Processing Systems, 30.