Large Language Model Tutor for Global Minority Languages

Introduction

Problem Description

The recent success of Large Language Models (LLMs) has driven a great deal of interest in applications ranging from homework help to global domination.

One of the most exciting possibilities for LLMs is their application to problems of knowledge accessibility.

Companies like DuoLingo and Khan Academy are leveraging LLMs and other forms of machine learning to provide a replacement for a human tutor. However, there is a dearth of learning resources available for global minority languages, that is, languages that are not among the languages most widely spoken.

One such language is Nepali. Nepali is spoken natively by 16 million people and is used as a second language by an additional 9 million, yet it is rarely available as an option for machine translation and LLMs do not cater to its speakers.

However, before effective tools can be made for the Nepali speaking population, it must also be possible for engineers and data scientists to learn Nepali.

Project Description

In this project I will endeavor to create an AI Nepali tutor. The tutor should be able to understand English language sentences from its student and provide Nepali translations of those sentences. Additionally, it should be able to generate novel sentences in English and provide Nepali translations of those sentences.

Further, Nepali uses the Devanagari script: a form of writing unfamiliar to many English speakers, especially in the west. Thus, the student should be able to provide an unfamiliar sentence in Nepali, written in this script, and receive a translation from the tutor.

For this project I will train a series of models. The first will translate English sentences to Nepali. The second will translate written Nepali into written English. Finally, the third will generate novel sentences in English.

This project presents a unique challenge. There is very little high-quality data available for training on the Nepali language. As such, I've put significant effort into optimizing my models to be as effective as possible with with very little data.

A github repo for this project can be found here:

https://github.com/billingsmoore/nepali-tutor

Note: Your computer may have trouble rendering the Devanagari script resulting in blanks where Nepali sentences should be. Viewing this notebook as a PDF should resolve that issue.

English to Nepali Translation

First, I will create a model to translate English sentences into Nepali sentences. To create this model, I drew on the Keras tutorial provided here:

https://keras.io/examples/nlp/neural_machine_translation_with_keras_nlp/

I've adapted the model from the tutorial to translate English into Nepali, rather than Spanish, and streamlined the code for simplicity and to meet my need for computational efficiency. Additionally, I've substantially altered the model in order to more fully optimize for the much, much smaller dataset available for the Nepali language.

The first step of this process is to import the necessary libraries.

Setup

In [ ]:
```
import pathlib
import random
import tensorflow as tf
from tensorflow import keras
import keras_nlp
import matplotlib.pyplot as plt
```

Next, I will establish the necessary constants for the model.

I will use a batch size of 4 for my data. This an EXTREMELY small batch size. I'm using such a small batch size in hopes of it helping to account for the extreme smallness of my dataset. This will lead to slower convergence. However, with a larger number of epochs and a careful choice of optimization algorithms (to be discussed later) we can get away with it.

I will train the model for 100 epochs. I had initially hoped to get away with fewer, however, with such small batch sizes, convergence takes longer.

I also establish a size for the vocabulary that the model will use and the dimensions for the model to expect from the data.

An interesting addition here is AUTOTUNE. tf.data.AUTOTUNE will automate optimization for training the model. This is extremely useful both for effectively utilizing computing resources, and for avoiding too much time lost to optimization tinkering.

In [ ]:
```
BATCH_SIZE = 4
EPOCHS = 100
MAX_SEQUENCE_LENGTH = 40
VOCAB_SIZE = 15000

EMBED_DIM = 256
INTERMEDIATE_DIM = 2048
NUM_HEADS = 8

AUTOTUNE = tf.data.AUTOTUNE
```

Importing and Exploring the Data

Now I will import the dataset that this model will be trained on. This data comes from Anki, a free, open-source flashcard program that is popular with language learners. It is available from this link:

https://www.manythings.org/anki/

I've then split the sentence pairs into two sets and set every English letter to be lowercase to avoid any confusion in the model. This is not necessary for Nepali because the Devanagari script does not use upper and lower cases.

In [ ]:
```
text_file = pathlib.Path('datasets/npi-eng/npi.txt')

with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    eng, nep = line.split("\t")[:2]
    eng = eng.lower()
    text_pairs.append((eng, nep))
```

The data comes in the form of numerous sentence pairs. First the sentence is given in English, then in Nepali. Each pair also has a source attribution, but that won't be necessary for the model. Below, I've printed some representative sentence pairs.

In [ ]:
```
for _ in range(5):
    print(random.choice(text_pairs))
```

```
('there are always exceptions.', 'त्यहाँ सधैं अपवादहरू छन्।')
('tom was killed in 2013.', 'टमको हत्या सन् २०१३ मा भएको थियो ।')
('we are classmates.', 'हामी सहपाठी हौं।')
('let tom do whatever he wants to do.', 'टमले जे गर्न चाहन्छ त्यो गर्न दिनुहोस्।')
('tom was quicker than mary.', 'टम मेरी भन्दा छिटो थियो।')
```

Now, we can split the sentence pairs into training, validation, and test sets. Notice that this dataset is quite small. This is one of the challenges of creating models for global minority languages. There is substantially less data to work with than if we were working with, for example, Spanish or French. As a result, I've allocated just 5% of the pairs to validation and testing respectively.

In [ ]:
```
random.shuffle(text_pairs)
num_val_samples = int(0.05 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples : num_train_samples + num_val_sampl
test_pairs = text_pairs[num_train_samples + num_val_samples:]

print(f"{len(text_pairs)} total pairs")
print(f"{len(train_pairs)} training pairs")
print(f"{len(val_pairs)} validation pairs")
print(f"{len(test_pairs)} test pairs")
```

```
1574 total pairs
1418 training pairs
78 validation pairs
78 test pairs
```

Creating the Tokenizer

The tokenizer will assign each unique word in the dataset a 'token' a unique number that allows the data to be treated numerically during model training. In order to do this, a "vocabulary" must first be created. This is a complete list of the unique English and Nepali words in the dataset.

Vocabulary

In [ ]:
```
def train_word_piece(text_samples, vocab_size, reserved_tokens):
    word_piece_ds = tf.data.Dataset.from_tensor_slices(text_samples)
    vocab = keras_nlp.tokenizers.compute_word_piece_vocabulary(
        word_piece_ds.batch(1000).prefetch(2),
        vocabulary_size=vocab_size,
        reserved_tokens=reserved_tokens,
```

```
    )

    return vocab
```

Tokenizing

Note that I've set aside some peculiar tokens. These correspond to whitespace,unknown characters, the beginnings and endings of sentences. I don't want the tokenizer to treat these things as words that need to be tokenized.

In [ ]:
```
reserved_tokens = ["[PAD]", "[UNK]", "[START]", "[END]"]

eng_samples = [text_pair[0] for text_pair in train_pairs]
eng_vocab = train_word_piece(eng_samples, VOCAB_SIZE, reserved_tokens)

nep_samples = [text_pair[1] for text_pair in train_pairs]
nep_vocab = train_word_piece(nep_samples, VOCAB_SIZE, reserved_tokens)
```

```
2023-08-18 10:59:12.371422: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.363718: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.364029: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.386559: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.387023: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.387300: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.714651: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.715049: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.715398: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-08-18 10:59:14.715709: I tensorflow/core/common_runtime/gpu/gpu_device.
cc:1639] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 10
075 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 4070, pci bus id: 000
0:01:00.0, compute capability: 8.9
```

Below we can see some example words from the dataset. Note that Nepali uses a distinct writing system that may not render correctly.

In [ ]:
```
print("English Tokens: ", eng_vocab[150:155])
print("Nepali Tokens: ", nep_vocab[200:205])
```

```
English Tokens:  ['much', 'old', 're', 'really', '##al']
Nepali Tokens:  ['किन', 'खान', 'तिमीलाई', 'पर्छ', 'भनेर']
```

Finally, we can tokenize the vocabularies.

In [ ]:
```
eng_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary=eng_vocab, lowercase=False
)

nep_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary=nep_vocab, lowercase=False
)
```

Data Preprocessing

Next, I will preprocess each batch of data. This consists of re-assembling the English-Nepali sentence pairs. Each sentence must be padded with the "[PAD]" whitespace token in order to make each sequence of tokens the same length. This is because the model expects inputs of a particular shape. Once the sentence has been padded to the appropriate length, a [START] token can be appended to the beginning and an [END] token appended to the end.

Finally, this assembled dataset can be split into training and validation sets.

In [ ]:
```
def eng_nep_preprocess_batch(eng, nep):
    batch_size = tf.shape(nep)[0]

    eng = eng_tokenizer(eng)
    nep = nep_tokenizer(nep)

    # pad eng to max_sequence_length
    eng_start_end_packer = keras_nlp.layers.StartEndPacker(
        sequence_length=MAX_SEQUENCE_LENGTH,
        pad_value = eng_tokenizer.token_to_id("[PAD]"),
    )

    eng = eng_start_end_packer(eng)

    # add special tokens [start] and [end] and pad nep
    nep_start_end_packer = keras_nlp.layers.StartEndPacker(
        sequence_length = MAX_SEQUENCE_LENGTH + 1,
        start_value = nep_tokenizer.token_to_id("[START]"),
        end_value = nep_tokenizer.token_to_id("[END]"),
        pad_value = nep_tokenizer.token_to_id("[PAD]")
    )

    nep = nep_start_end_packer(nep)

    return (
        {
        "encoder_inputs": eng,
        "decoder_inputs": nep[:, :-1]
        },
```

```
        nep[:, 1:],
    )

def make_dataset(pairs):
    eng_texts, nep_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    nep_texts = list(nep_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, nep_texts))
    dataset=dataset.batch(BATCH_SIZE)
    dataset = dataset.map(eng_nep_preprocess_batch, num_parallel_calls=AUTOT
    return dataset.shuffle(2048).prefetch(16).cache()

eng_nep_train_ds = make_dataset(train_pairs)
eng_nep_val_ds = make_dataset(val_pairs)
```

Creating the Model

Now it's time to build the model itself. This model is an Autoencoder, which consists of an encoder and a decoder.

The encoder input layer takes in a set of tokenized inputs. These inputs are then passed to a layer that accounts for the number assigned to the token as well as the position of that token in the sentence. The next layer is a typical dense Encoder layer.

The decoder takes in a set of tokenized inputs from the Nepali dataset and passes them to a layer that will account for the token number and position of the token in those sentences. This is then passed to a typical dense Decoder layer.

Both the Encoder and Decoder layers are helpfully provided out-of-the-box by Keras.

In [ ]:
```
encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_inp

x = keras_nlp.layers.TokenAndPositionEmbedding(
    vocabulary_size=VOCAB_SIZE,
    sequence_length = MAX_SEQUENCE_LENGTH,
    embedding_dim=EMBED_DIM,
    mask_zero=True,
)(encoder_inputs)

encoder_outputs = keras_nlp.layers.TransformerEncoder(
    intermediate_dim = INTERMEDIATE_DIM, num_heads=NUM_HEADS
)(inputs=x)
encoder = keras.Model(encoder_inputs, encoder_outputs)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="decoder_inp
encoded_seq_inputs = keras.Input(shape=(None, EMBED_DIM), name="decoder_stat

x = keras_nlp.layers.TokenAndPositionEmbedding(
    vocabulary_size=VOCAB_SIZE,
    sequence_length=MAX_SEQUENCE_LENGTH,
    embedding_dim=EMBED_DIM,
    mask_zero=True,
)(decoder_inputs)
```

```
x = keras_nlp.layers.TransformerDecoder(
    intermediate_dim=INTERMEDIATE_DIM, num_heads=NUM_HEADS
)(decoder_sequence=x, encoder_sequence=encoded_seq_inputs)
x = keras.layers.Dropout(0.5)(x)
decoder_outputs = keras.layers.Dense(VOCAB_SIZE, activation="softmax")(x)
decoder = keras.Model(
    [
        decoder_inputs,
        encoded_seq_inputs
    ],
    decoder_outputs,
)
decoder_outputs = decoder([decoder_inputs, encoder_outputs])

eng_nep_translator = keras.Model(
    [encoder_inputs, decoder_inputs],
    decoder_outputs,
    name="eng_nep_translator",
)
```

Model Summary

In [ ]:
```
eng_nep_translator.summary()
```

```
Model: "eng_nep_translator"
__________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connecte
d to
==========================================================================================
 encoder_inputs (InputLayer  [(None, None)]            0         []
 )


__________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connecte
d to
==========================================================================================
 encoder_inputs (InputLayer  [(None, None)]            0         []
 )

 token_and_position_embeddi  (None, None, 256)         3850240   ['encode
r_inputs[0][0]']
 ng (TokenAndPositionEmbedd
 ing)

 decoder_inputs (InputLayer  [(None, None)]            0         []
 )

 transformer_encoder (Trans  (None, None, 256)         1315072   ['token_
and_position_embedding
 formerEncoder)                                                   [0][0]']

 model_1 (Functional)        (None, None, 15000)       9283992   ['decode
r_inputs[0][0]',
                                                                  'transf
ormer_encoder[0][0]']

==========================================================================================
Total params: 14449304 (55.12 MB)
Trainable params: 14449304 (55.12 MB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________
```

Compilation

Now, I've compiled the model.

Of note here is the choice of optimization algorith. I have used RMSProp. RMSProp is similar to Adagrad, which we studied in class, and as a result it converges much more quickly than, say, SGD. However, it is less susceptible to vanishing gradients. This is perfect for our small dataset with small batch sizes.

The loss function is Sparse Categorical Crossentropy. Not every word appears in every

sentence so the data for most natural language related tasks is necessarily sparse.

In [ ]:
```
eng_nep_translator.compile(
    "rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Fitting the Model

In [ ]:
```
eng_nep_history = eng_nep_translator.fit(
    eng_nep_train_ds,
    epochs=100,
    validation_data=eng_nep_val_ds
    )
```

```
Epoch 1/100
23/23 [==============================] - 15s 301ms/step - loss: 5.7427 - acc
uracy: 0.1268 - val_loss: 4.6218 - val_accuracy: 0.2142
Epoch 2/100
23/23 [==============================] - 0s 21ms/step - loss: 4.3512 - accur
acy: 0.2179 - val_loss: 4.2112 - val_accuracy: 0.2256
Epoch 3/100
23/23 [==============================] - 1s 24ms/step - loss: 3.9231 - accur
acy: 0.2553 - val_loss: 3.8240 - val_accuracy: 0.2549
Epoch 4/100
23/23 [==============================] - 1s 22ms/step - loss: 3.6029 - accur
acy: 0.2820 - val_loss: 3.5899 - val_accuracy: 0.2769
Epoch 5/100
23/23 [==============================] - 1s 22ms/step - loss: 3.3899 - accur
acy: 0.2981 - val_loss: 3.5028 - val_accuracy: 0.2834
Epoch 6/100
23/23 [==============================] - 1s 22ms/step - loss: 3.2422 - accur
acy: 0.3088 - val_loss: 3.4117 - val_accuracy: 0.2818
Epoch 7/100
23/23 [==============================] - 0s 20ms/step - loss: 3.1218 - accur
acy: 0.3177 - val_loss: 3.3454 - val_accuracy: 0.2956
Epoch 8/100
23/23 [==============================] - 0s 21ms/step - loss: 3.0220 - accur
acy: 0.3251 - val_loss: 3.2250 - val_accuracy: 0.3119
Epoch 9/100
23/23 [==============================] - 1s 24ms/step - loss: 2.9113 - accur
acy: 0.3407 - val_loss: 3.2453 - val_accuracy: 0.3111
Epoch 10/100
23/23 [==============================] - 0s 20ms/step - loss: 2.8396 - accur
acy: 0.3457 - val_loss: 3.2248 - val_accuracy: 0.3200
Epoch 11/100
23/23 [==============================] - 0s 21ms/step - loss: 2.7460 - accur
acy: 0.3575 - val_loss: 3.0920 - val_accuracy: 0.3217
Epoch 12/100
23/23 [==============================] - 1s 25ms/step - loss: 2.6709 - accur
acy: 0.3706 - val_loss: 3.0824 - val_accuracy: 0.3135
Epoch 13/100
23/23 [==============================] - 0s 20ms/step - loss: 2.6100 - accur
acy: 0.3795 - val_loss: 3.0974 - val_accuracy: 0.3331
Epoch 14/100
23/23 [==============================] - 0s 21ms/step - loss: 2.5298 - accur
acy: 0.3945 - val_loss: 3.0074 - val_accuracy: 0.3388
Epoch 15/100
23/23 [==============================] - 1s 24ms/step - loss: 2.4643 - accur
acy: 0.4047 - val_loss: 2.9844 - val_accuracy: 0.3436
Epoch 16/100
23/23 [==============================] - 0s 20ms/step - loss: 2.3783 - accur
acy: 0.4193 - val_loss: 3.0206 - val_accuracy: 0.3445
Epoch 17/100
23/23 [==============================] - 0s 20ms/step - loss: 2.3049 - accur
acy: 0.4322 - val_loss: 2.9466 - val_accuracy: 0.3575
Epoch 18/100
23/23 [==============================] - 1s 23ms/step - loss: 2.2295 - accur
acy: 0.4487 - val_loss: 2.9362 - val_accuracy: 0.3648
Epoch 19/100
23/23 [==============================] - 0s 21ms/step - loss: 2.1355 - accur
```

```
acy: 0.4660 - val_loss: 2.8849 - val_accuracy: 0.3599
Epoch 20/100
23/23 [==============================] - 0s 20ms/step - loss: 2.0511 - accur
acy: 0.4803 - val_loss: 2.9393 - val_accuracy: 0.3640
Epoch 21/100
23/23 [==============================] - 1s 25ms/step - loss: 1.9651 - accur
acy: 0.5017 - val_loss: 2.9228 - val_accuracy: 0.3770
Epoch 22/100
23/23 [==============================] - 0s 20ms/step - loss: 1.8585 - accur
acy: 0.5265 - val_loss: 2.8914 - val_accuracy: 0.3884
Epoch 23/100
23/23 [==============================] - 0s 21ms/step - loss: 1.7961 - accur
acy: 0.5374 - val_loss: 2.8964 - val_accuracy: 0.3958
Epoch 24/100
23/23 [==============================] - 1s 29ms/step - loss: 1.6807 - accur
acy: 0.5622 - val_loss: 2.8951 - val_accuracy: 0.4055
Epoch 25/100
23/23 [==============================] - 0s 22ms/step - loss: 1.6087 - accur
acy: 0.5791 - val_loss: 2.9280 - val_accuracy: 0.4047
Epoch 26/100
23/23 [==============================] - 0s 20ms/step - loss: 1.4642 - accur
acy: 0.6144 - val_loss: 2.9629 - val_accuracy: 0.3974
Epoch 27/100
23/23 [==============================] - 1s 23ms/step - loss: 1.3917 - accur
acy: 0.6302 - val_loss: 2.9667 - val_accuracy: 0.4169
Epoch 28/100
23/23 [==============================] - 0s 21ms/step - loss: 1.2891 - accur
acy: 0.6569 - val_loss: 3.0268 - val_accuracy: 0.4153
Epoch 29/100
23/23 [==============================] - 0s 21ms/step - loss: 1.1906 - accur
acy: 0.6793 - val_loss: 3.0415 - val_accuracy: 0.4259
Epoch 30/100
23/23 [==============================] - 1s 24ms/step - loss: 1.0887 - accur
acy: 0.7080 - val_loss: 3.0412 - val_accuracy: 0.4169
Epoch 31/100
23/23 [==============================] - 0s 21ms/step - loss: 1.0141 - accur
acy: 0.7255 - val_loss: 3.1149 - val_accuracy: 0.4235
Epoch 32/100
23/23 [==============================] - 0s 20ms/step - loss: 0.9232 - accur
acy: 0.7504 - val_loss: 3.1392 - val_accuracy: 0.4137
Epoch 33/100
23/23 [==============================] - 0s 21ms/step - loss: 0.8118 - accur
acy: 0.7783 - val_loss: 3.1473 - val_accuracy: 0.4340
Epoch 34/100
23/23 [==============================] - 0s 21ms/step - loss: 0.7574 - accur
acy: 0.7898 - val_loss: 3.2356 - val_accuracy: 0.4267
Epoch 35/100
23/23 [==============================] - 1s 23ms/step - loss: 0.6617 - accur
acy: 0.8183 - val_loss: 3.2747 - val_accuracy: 0.4316
Epoch 36/100
23/23 [==============================] - 0s 20ms/step - loss: 0.6310 - accur
acy: 0.8265 - val_loss: 3.3351 - val_accuracy: 0.4373
Epoch 37/100
23/23 [==============================] - 0s 20ms/step - loss: 0.5566 - accur
acy: 0.8472 - val_loss: 3.3859 - val_accuracy: 0.4324
Epoch 38/100
```

```
23/23 [==============================] - 1s 23ms/step - loss: 0.4769 - accur
acy: 0.8699 - val_loss: 3.4272 - val_accuracy: 0.4283
Epoch 39/100
23/23 [==============================] - 0s 21ms/step - loss: 0.4445 - accur
acy: 0.8798 - val_loss: 3.4757 - val_accuracy: 0.4454
Epoch 40/100
23/23 [==============================] - 0s 19ms/step - loss: 0.3955 - accur
acy: 0.8925 - val_loss: 3.5336 - val_accuracy: 0.4292
Epoch 41/100
23/23 [==============================] - 1s 23ms/step - loss: 0.3527 - accur
acy: 0.9047 - val_loss: 3.5205 - val_accuracy: 0.4389
Epoch 42/100
23/23 [==============================] - 1s 22ms/step - loss: 0.3362 - accur
acy: 0.9081 - val_loss: 3.6622 - val_accuracy: 0.4332
Epoch 43/100
23/23 [==============================] - 1s 22ms/step - loss: 0.3422 - accur
acy: 0.9060 - val_loss: 3.6460 - val_accuracy: 0.4430
Epoch 44/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2685 - accur
acy: 0.9253 - val_loss: 3.7569 - val_accuracy: 0.4300
Epoch 45/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2509 - accur
acy: 0.9306 - val_loss: 3.8573 - val_accuracy: 0.4251
Epoch 46/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2338 - accur
acy: 0.9372 - val_loss: 3.8833 - val_accuracy: 0.4381
Epoch 47/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2231 - accur
acy: 0.9386 - val_loss: 3.8998 - val_accuracy: 0.4397
Epoch 48/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2155 - accur
acy: 0.9408 - val_loss: 3.9011 - val_accuracy: 0.4438
Epoch 49/100
23/23 [==============================] - 0s 20ms/step - loss: 0.1991 - accur
acy: 0.9461 - val_loss: 3.9741 - val_accuracy: 0.4422
Epoch 50/100
23/23 [==============================] - 1s 23ms/step - loss: 0.1889 - accur
acy: 0.9498 - val_loss: 4.0333 - val_accuracy: 0.4316
Epoch 51/100
23/23 [==============================] - 0s 21ms/step - loss: 0.1833 - accur
acy: 0.9500 - val_loss: 4.0515 - val_accuracy: 0.4349
Epoch 52/100
23/23 [==============================] - 0s 20ms/step - loss: 0.1658 - accur
acy: 0.9549 - val_loss: 4.0739 - val_accuracy: 0.4340
Epoch 53/100
23/23 [==============================] - 1s 25ms/step - loss: 0.1594 - accur
acy: 0.9573 - val_loss: 4.1424 - val_accuracy: 0.4332
Epoch 54/100
23/23 [==============================] - 0s 21ms/step - loss: 0.1616 - accur
acy: 0.9560 - val_loss: 4.1376 - val_accuracy: 0.4454
Epoch 55/100
23/23 [==============================] - 0s 21ms/step - loss: 0.1418 - accur
acy: 0.9612 - val_loss: 4.1774 - val_accuracy: 0.4389
Epoch 56/100
23/23 [==============================] - 1s 23ms/step - loss: 0.1401 - accur
acy: 0.9647 - val_loss: 4.1605 - val_accuracy: 0.4446
```

```
Epoch 57/100
23/23 [==============================] - 0s 20ms/step - loss: 0.1364 - accur
acy: 0.9643 - val_loss: 4.2501 - val_accuracy: 0.4275
Epoch 58/100
23/23 [==============================] - 0s 21ms/step - loss: 0.1316 - accur
acy: 0.9652 - val_loss: 4.2439 - val_accuracy: 0.4536
Epoch 59/100
23/23 [==============================] - 1s 23ms/step - loss: 0.1282 - accur
acy: 0.9654 - val_loss: 4.2698 - val_accuracy: 0.4463
Epoch 60/100
23/23 [==============================] - 0s 20ms/step - loss: 0.1233 - accur
acy: 0.9675 - val_loss: 4.3258 - val_accuracy: 0.4414
Epoch 61/100
23/23 [==============================] - 0s 20ms/step - loss: 0.1196 - accur
acy: 0.9679 - val_loss: 4.3740 - val_accuracy: 0.4454
Epoch 62/100
23/23 [==============================] - 1s 22ms/step - loss: 0.1154 - accur
acy: 0.9695 - val_loss: 4.3361 - val_accuracy: 0.4389
Epoch 63/100
23/23 [==============================] - 0s 21ms/step - loss: 0.1115 - accur
acy: 0.9700 - val_loss: 4.4678 - val_accuracy: 0.4365
Epoch 64/100
23/23 [==============================] - 0s 20ms/step - loss: 0.1155 - accur
acy: 0.9684 - val_loss: 4.4577 - val_accuracy: 0.4324
Epoch 65/100
23/23 [==============================] - 1s 23ms/step - loss: 0.1036 - accur
acy: 0.9726 - val_loss: 4.4356 - val_accuracy: 0.4349
Epoch 66/100
23/23 [==============================] - 0s 21ms/step - loss: 0.1136 - accur
acy: 0.9692 - val_loss: 4.4343 - val_accuracy: 0.4340
Epoch 67/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0921 - accur
acy: 0.9760 - val_loss: 4.4924 - val_accuracy: 0.4243
Epoch 68/100
23/23 [==============================] - 1s 25ms/step - loss: 0.1068 - accur
acy: 0.9703 - val_loss: 4.5033 - val_accuracy: 0.4438
Epoch 69/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0934 - accur
acy: 0.9755 - val_loss: 4.5548 - val_accuracy: 0.4389
Epoch 70/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0907 - accur
acy: 0.9762 - val_loss: 4.4792 - val_accuracy: 0.4332
Epoch 71/100
23/23 [==============================] - 1s 24ms/step - loss: 0.0843 - accur
acy: 0.9789 - val_loss: 4.5505 - val_accuracy: 0.4414
Epoch 72/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0963 - accur
acy: 0.9757 - val_loss: 4.5601 - val_accuracy: 0.4357
Epoch 73/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0932 - accur
acy: 0.9745 - val_loss: 4.6776 - val_accuracy: 0.4316
Epoch 74/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0986 - accur
acy: 0.9733 - val_loss: 4.5429 - val_accuracy: 0.4324
Epoch 75/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0865 - accur
```

```
acy: 0.9772 - val_loss: 4.5644 - val_accuracy: 0.4389
Epoch 76/100
23/23 [==============================] - 1s 22ms/step - loss: 0.0760 - accur
acy: 0.9817 - val_loss: 4.5976 - val_accuracy: 0.4406
Epoch 77/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0812 - accur
acy: 0.9795 - val_loss: 4.6826 - val_accuracy: 0.4397
Epoch 78/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0825 - accur
acy: 0.9785 - val_loss: 4.6726 - val_accuracy: 0.4544
Epoch 79/100
23/23 [==============================] - 1s 25ms/step - loss: 0.0804 - accur
acy: 0.9801 - val_loss: 4.7449 - val_accuracy: 0.4454
Epoch 80/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0756 - accur
acy: 0.9796 - val_loss: 4.8342 - val_accuracy: 0.4389
Epoch 81/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0810 - accur
acy: 0.9773 - val_loss: 4.7415 - val_accuracy: 0.4332
Epoch 82/100
23/23 [==============================] - 1s 24ms/step - loss: 0.0691 - accur
acy: 0.9818 - val_loss: 4.7434 - val_accuracy: 0.4308
Epoch 83/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0698 - accur
acy: 0.9819 - val_loss: 4.8233 - val_accuracy: 0.4430
Epoch 84/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0779 - accur
acy: 0.9788 - val_loss: 4.7662 - val_accuracy: 0.4349
Epoch 85/100
23/23 [==============================] - 1s 24ms/step - loss: 0.0780 - accur
acy: 0.9792 - val_loss: 4.8281 - val_accuracy: 0.4349
Epoch 86/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0723 - accur
acy: 0.9808 - val_loss: 4.7724 - val_accuracy: 0.4471
Epoch 87/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0709 - accur
acy: 0.9810 - val_loss: 4.8149 - val_accuracy: 0.4430
Epoch 88/100
23/23 [==============================] - 1s 23ms/step - loss: 0.0728 - accur
acy: 0.9800 - val_loss: 4.7687 - val_accuracy: 0.4487
Epoch 89/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0648 - accur
acy: 0.9834 - val_loss: 4.8348 - val_accuracy: 0.4267
Epoch 90/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0703 - accur
acy: 0.9818 - val_loss: 4.8870 - val_accuracy: 0.4389
Epoch 91/100
23/23 [==============================] - 1s 24ms/step - loss: 0.0675 - accur
acy: 0.9815 - val_loss: 4.8855 - val_accuracy: 0.4365
Epoch 92/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0644 - accur
acy: 0.9830 - val_loss: 4.8516 - val_accuracy: 0.4430
Epoch 93/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0672 - accur
acy: 0.9820 - val_loss: 4.8644 - val_accuracy: 0.4422
Epoch 94/100
```

```
23/23 [==============================] - 1s 23ms/step - loss: 0.0683 - accur
acy: 0.9816 - val_loss: 4.8650 - val_accuracy: 0.4397
Epoch 95/100
23/23 [==============================] - 0s 22ms/step - loss: 0.0673 - accur
acy: 0.9826 - val_loss: 4.8566 - val_accuracy: 0.4300
Epoch 96/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0629 - accur
acy: 0.9832 - val_loss: 4.8677 - val_accuracy: 0.4422
Epoch 97/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0682 - accur
acy: 0.9816 - val_loss: 4.8726 - val_accuracy: 0.4463
Epoch 98/100
23/23 [==============================] - 0s 20ms/step - loss: 0.0583 - accur
acy: 0.9837 - val_loss: 4.7683 - val_accuracy: 0.4430
Epoch 99/100
23/23 [==============================] - 1s 23ms/step - loss: 0.0597 - accur
acy: 0.9852 - val_loss: 4.7806 - val_accuracy: 0.4389
Epoch 100/100
23/23 [==============================] - 0s 21ms/step - loss: 0.0567 - accur
acy: 0.9854 - val_loss: 4.9264 - val_accuracy: 0.4397
```

To avoid training and retraining the model, I'll now save this model with these results.

In [ ]:
```
#eng_nep_translator.save('models/eng-nep-translator.keras')
```

Now, for future testing, I can reopen the model.

In [ ]:
```
eng_nep_translator = tf.keras.models.load_model('models/eng-nep-translator.k
```

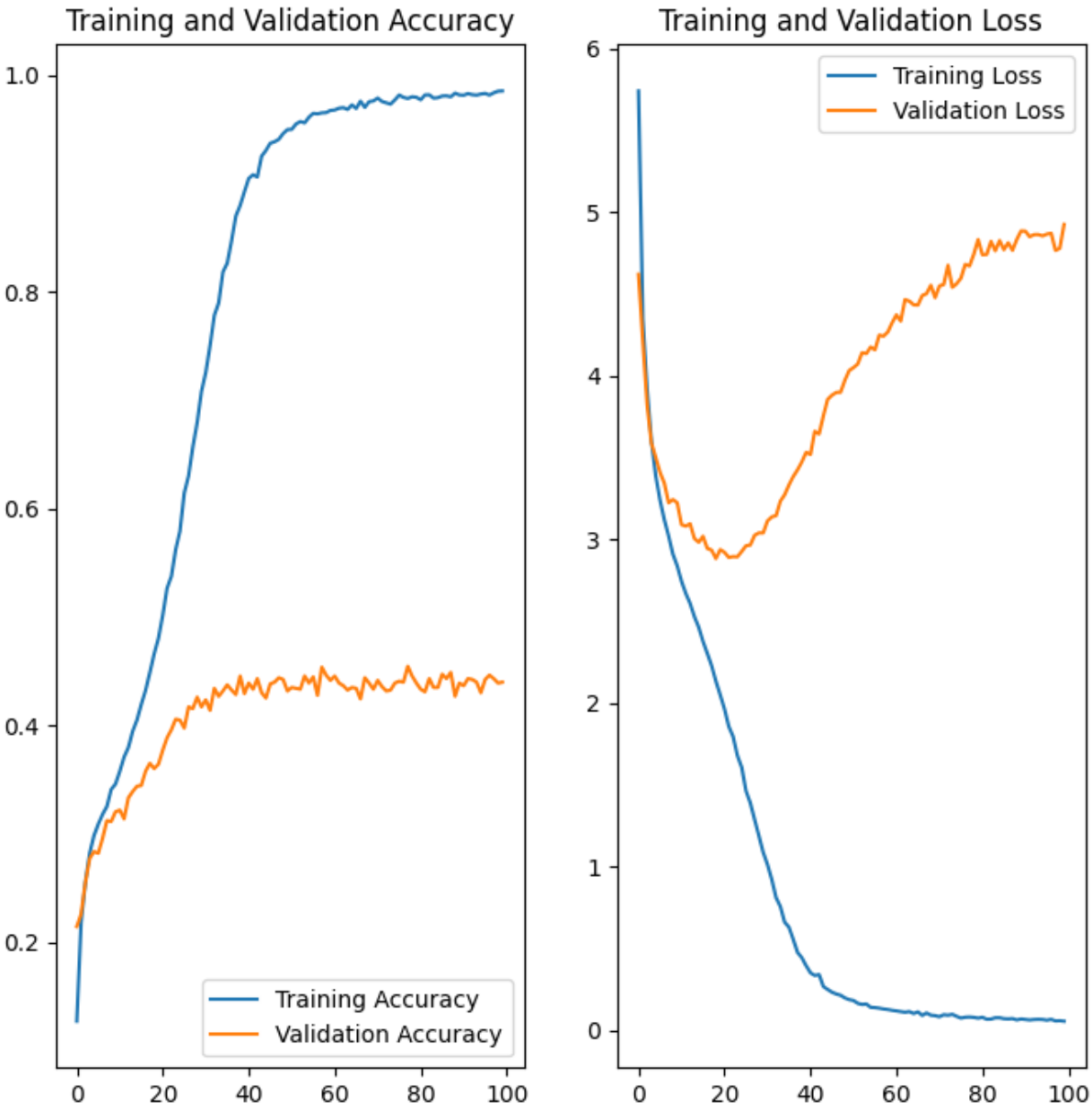
Visualizing the Training Results

Below, we can see how the loss and accuracy evolved over the course of training. Here we can really see how difficult it is to make effective generative tools from small datasets. Even as the model's accuracy improves substantially on the training set, the accuracy on the validation data remains unacceptably low.

The loss on the validation data also never decreases, instead getting worse as time goes on.

In [ ]:
```
acc = eng_nep_history.history['accuracy']
val_acc = eng_nep_history.history['val_accuracy']

loss = eng_nep_history.history['loss']
val_loss = eng_nep_history.history['val_loss']

epochs_range = range(100)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
```

```
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



Decoding Translated Sentences

Even if the translations are perfect, the outputs of our model are not meaningful sentences. The model only outputs numerical tokens. In order to turn these into something that a human can read they need to be decoded.

Below is a function to decode these translated sentences. This function takes in an English sentence, runs it through our translator model then works its way through the output of

the model, converting the output into words in Nepali using our tokenizers.

Part of decoding the sequence is sampling the probabilities of tokens that should follow the existing translation. The sampler is the algorithm that is used to select that next work or token. Here I've used the Greedy sampler which simply finds the highest likelihood next word and adds it to the translated sentence. It is computationally inexpensive and because the outputs are pretty short we don't need to worry about the Greedy sampler outputting long, repetitive sentences that don't make much sense, which can be an issue with the algorithm.

In [ ]:
```
def eng_nep_translate(input_sentences):
    batch_size = tf.shape(input_sentences)[0]

    encoder_input_tokens = eng_tokenizer(input_sentences).to_tensor(
        shape=(None, MAX_SEQUENCE_LENGTH)
    )

    def next(prompt, cache, index):
        logits = eng_nep_translator([encoder_input_tokens, prompt])[:, index
        hidden_states = None
        return logits, hidden_states, cache

    length = 40
    start = tf.fill((batch_size, 1), nep_tokenizer.token_to_id("[START]"))
    pad = tf.fill((batch_size, length - 1), nep_tokenizer.token_to_id("[PAD]
    prompt = tf.concat((start, pad), axis=-1)

    generated_tokens = keras_nlp.samplers.GreedySampler()(
        next,
        prompt,
        end_token_id=nep_tokenizer.token_to_id("[END]"),
        index=1
    )
    generated_sentences = nep_tokenizer.detokenize(generated_tokens)
    return generated_sentences
```

Example Translations

Now, let's look at some example translations from the model.

In [ ]:
```
test_eng_texts = [pair[0] for pair in test_pairs]
for i in range(5):
    input_sentence = random.choice(test_eng_texts)
    translated = eng_nep_translate(tf.constant([input_sentence]))
    translated = translated.numpy()[0].decode("utf-8")
    translated = (
        translated.replace("[PAD]", "")
        .replace("[START]", "")
        .replace("[END]", "")
        .strip()
    )
```

```
    print(f"** Example {i} **")
    print(input_sentence)
    print(translated)
    print()
```

```
2023-08-18 10:59:35.830764: I tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:606] TensorFloat-32 will be used for the matrix multiplication. This will only be logged once.
2023-08-18 10:59:36.980078: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x1051b930 initialized for platform CUDA (this does not guarantee that XLA will be used). Devices:
2023-08-18 10:59:36.980152: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): NVIDIA GeForce RTX 4070, Compute Capability 8.9
2023-08-18 10:59:38.690555: I tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:432] Loaded cuDNN version 8904
2023-08-18 10:59:40.544634: I tensorflow/tsl/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-08-18 10:59:41.517654: I ./tensorflow/compiler/jit/device_compiler.h:186] Compiled cluster using XLA!  This line is logged at most once for the lifetime of the process.
** Example 0 **
i didn't want to go, but i did.
ो ठ चाहन्छौं एक अहिलेला ,ब छिन्ौं ड हुँ हुनुहुन्छान थाहाको

** Example 1 **
i know that tom is in bed.
मैलु दहरू ौलाई काम थाहाको

** Example 2 **
you won't see me again.
ह मैले ी ा अहिले टमलाई छिन्ो ्ईस थाहा , मैलेलेको

** Example 3 **
when was the last time you cried?
गर्न0 हामी ढसदैनिएको किस हुँै ?

** Example 4 **
i'm pretty sure tom's happy in boston.
मैले ु चाहन्छुरही ढससँगससँगिएको कि थ  एक ठदैन तपाई दको
```

Nepali to English Translation

Now that the English to Nepali Translator is finished, it's time to reverse the process.

There's no need to repeat the setup stages nor the importing and preprocessing of data, since it can all be reused from the first translator. We can simply refactor our preprocess_batch function to reverse which language is the input and which language is the output.

In [ ]:
```
def nep_eng_preprocess_batch(eng, nep):
```

```
    eng = eng_tokenizer(eng)
    nep = nep_tokenizer(nep)

    # pad eng to max_sequence_length
    eng_start_end_packer = keras_nlp.layers.StartEndPacker(
        sequence_length=MAX_SEQUENCE_LENGTH+1,
        pad_value = eng_tokenizer.token_to_id("[PAD]"),
    )

    eng = eng_start_end_packer(eng)

    # add special tokens [start] and [end] and pad nep
    nep_start_end_packer = keras_nlp.layers.StartEndPacker(
        sequence_length = MAX_SEQUENCE_LENGTH,
        start_value = nep_tokenizer.token_to_id("[START]"),
        end_value = nep_tokenizer.token_to_id("[END]"),
        pad_value = nep_tokenizer.token_to_id("[PAD]")
    )

    nep = nep_start_end_packer(nep)

    return (
        {
        "encoder_inputs": nep,
        "decoder_inputs": eng[:, :-1]
        },
        eng[:, 1:],
    )

def make_dataset(pairs):
    eng_texts, nep_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    nep_texts = list(nep_texts)
    dataset = tf.data.Dataset.from_tensor_slices((nep_texts, eng_texts))
    dataset=dataset.batch(BATCH_SIZE)
    dataset = dataset.map(nep_eng_preprocess_batch, num_parallel_calls=AUTOT
    return dataset.shuffle(2048).prefetch(16).cache()

nep_eng_train_ds = make_dataset(train_pairs)
nep_eng_val_ds = make_dataset(val_pairs)
```

Creating the Model

We can reuse the same model architecture from above.

In [ ]:
```
nep_eng_translator = keras.Model(
    [encoder_inputs, decoder_inputs],
    decoder_outputs,
    name="nep_eng_translator",
)
```

Model Summary

In [ ]:
```
nep_eng_translator.summary()
```

```
Model: "nep_eng_translator"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 encoder_inputs (InputLayer  [(None, None)]            0         []
 )


_________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 encoder_inputs (InputLayer  [(None, None)]            0         []
 )

 token_and_position_embeddi  (None, None, 256)         3850240   ['encoder_inputs[0][0]']
 ng (TokenAndPositionEmbedding)

 decoder_inputs (InputLayer  [(None, None)]            0         []
 )

 transformer_encoder (Trans  (None, None, 256)         1315072   ['token_and_position_embedding
 formerEncoder)                                                  [0][0]']

 model_1 (Functional)        (None, None, 15000)       9283992   ['decoder_inputs[0][0]',
                                                                  'transformer_encoder[0][0]']

==================================================================================================
Total params: 14449304 (55.12 MB)
Trainable params: 14449304 (55.12 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Compilation

Here I've used the same loss and optimization algorithms as the other translator.

In [ ]:
```
nep_eng_translator.compile(
    "rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Fitting the Model

In [ ]:
```
nep_eng_history = nep_eng_translator.fit(
    nep_eng_train_ds,
    epochs=100,
    validation_data=nep_eng_val_ds
    )
```

```
Epoch 1/100
23/23 [==============================] - 12s 140ms/step - loss: 1.8171 - acc
uracy: 0.7730 - val_loss: 0.6445 - val_accuracy: 0.7848
Epoch 2/100
23/23 [==============================] - 1s 24ms/step - loss: 0.5696 - accur
acy: 0.8187 - val_loss: 0.5252 - val_accuracy: 0.8135
Epoch 3/100
23/23 [==============================] - 1s 22ms/step - loss: 0.4751 - accur
acy: 0.8355 - val_loss: 0.4863 - val_accuracy: 0.8463
Epoch 4/100
23/23 [==============================] - 0s 21ms/step - loss: 0.4529 - accur
acy: 0.8464 - val_loss: 0.4911 - val_accuracy: 0.8135
Epoch 5/100
23/23 [==============================] - 1s 23ms/step - loss: 0.4292 - accur
acy: 0.8491 - val_loss: 0.4543 - val_accuracy: 0.8484
Epoch 6/100
23/23 [==============================] - 0s 20ms/step - loss: 0.4058 - accur
acy: 0.8572 - val_loss: 0.4350 - val_accuracy: 0.8525
Epoch 7/100
23/23 [==============================] - 0s 21ms/step - loss: 0.3929 - accur
acy: 0.8598 - val_loss: 0.3791 - val_accuracy: 0.8607
Epoch 8/100
23/23 [==============================] - 1s 26ms/step - loss: 0.3552 - accur
acy: 0.8600 - val_loss: 0.3576 - val_accuracy: 0.8689
Epoch 9/100
23/23 [==============================] - 0s 20ms/step - loss: 0.3352 - accur
acy: 0.8639 - val_loss: 0.3597 - val_accuracy: 0.8709
Epoch 10/100
23/23 [==============================] - 1s 24ms/step - loss: 0.3233 - accur
acy: 0.8699 - val_loss: 0.3577 - val_accuracy: 0.8668
Epoch 11/100
23/23 [==============================] - 0s 21ms/step - loss: 0.3192 - accur
acy: 0.8716 - val_loss: 0.3513 - val_accuracy: 0.8586
Epoch 12/100
23/23 [==============================] - 0s 20ms/step - loss: 0.3083 - accur
acy: 0.8706 - val_loss: 0.3578 - val_accuracy: 0.8709
Epoch 13/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2982 - accur
acy: 0.8702 - val_loss: 0.3527 - val_accuracy: 0.8709
Epoch 14/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2976 - accur
acy: 0.8712 - val_loss: 0.3507 - val_accuracy: 0.8668
Epoch 15/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2925 - accur
acy: 0.8716 - val_loss: 0.3481 - val_accuracy: 0.8566
Epoch 16/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2849 - accur
acy: 0.8758 - val_loss: 0.3461 - val_accuracy: 0.8566
Epoch 17/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2833 - accur
acy: 0.8780 - val_loss: 0.3553 - val_accuracy: 0.8463
Epoch 18/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2785 - accur
acy: 0.8763 - val_loss: 0.3430 - val_accuracy: 0.8607
Epoch 19/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2761 - accur
```

```
acy: 0.8756 - val_loss: 0.3332 - val_accuracy: 0.8730
Epoch 20/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2760 - accur
acy: 0.8749 - val_loss: 0.3532 - val_accuracy: 0.8607
Epoch 21/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2706 - accur
acy: 0.8786 - val_loss: 0.3422 - val_accuracy: 0.8730
Epoch 22/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2705 - accur
acy: 0.8773 - val_loss: 0.3471 - val_accuracy: 0.8689
Epoch 23/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2681 - accur
acy: 0.8790 - val_loss: 0.3649 - val_accuracy: 0.8566
Epoch 24/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2703 - accur
acy: 0.8772 - val_loss: 0.3464 - val_accuracy: 0.8668
Epoch 25/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2663 - accur
acy: 0.8780 - val_loss: 0.3523 - val_accuracy: 0.8627
Epoch 26/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2671 - accur
acy: 0.8775 - val_loss: 0.3432 - val_accuracy: 0.8730
Epoch 27/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2639 - accur
acy: 0.8801 - val_loss: 0.3502 - val_accuracy: 0.8689
Epoch 28/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2625 - accur
acy: 0.8791 - val_loss: 0.3513 - val_accuracy: 0.8668
Epoch 29/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2583 - accur
acy: 0.8815 - val_loss: 0.3720 - val_accuracy: 0.8648
Epoch 30/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2596 - accur
acy: 0.8781 - val_loss: 0.3625 - val_accuracy: 0.8668
Epoch 31/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2626 - accur
acy: 0.8778 - val_loss: 0.3449 - val_accuracy: 0.8607
Epoch 32/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2546 - accur
acy: 0.8827 - val_loss: 0.3527 - val_accuracy: 0.8648
Epoch 33/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2533 - accur
acy: 0.8825 - val_loss: 0.3646 - val_accuracy: 0.8525
Epoch 34/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2576 - accur
acy: 0.8795 - val_loss: 0.3614 - val_accuracy: 0.8668
Epoch 35/100
23/23 [==============================] - 1s 25ms/step - loss: 0.2573 - accur
acy: 0.8829 - val_loss: 0.3609 - val_accuracy: 0.8648
Epoch 36/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2550 - accur
acy: 0.8852 - val_loss: 0.3523 - val_accuracy: 0.8607
Epoch 37/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2502 - accur
acy: 0.8832 - val_loss: 0.3474 - val_accuracy: 0.8627
Epoch 38/100
```

```
23/23 [==============================] - 0s 21ms/step - loss: 0.2486 - accur
acy: 0.8842 - val_loss: 0.3473 - val_accuracy: 0.8668
Epoch 39/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2467 - accur
acy: 0.8834 - val_loss: 0.3717 - val_accuracy: 0.8586
Epoch 40/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2485 - accur
acy: 0.8831 - val_loss: 0.3653 - val_accuracy: 0.8627
Epoch 41/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2448 - accur
acy: 0.8855 - val_loss: 0.3647 - val_accuracy: 0.8627
Epoch 42/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2445 - accur
acy: 0.8860 - val_loss: 0.3688 - val_accuracy: 0.8504
Epoch 43/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2483 - accur
acy: 0.8850 - val_loss: 0.3493 - val_accuracy: 0.8648
Epoch 44/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2548 - accur
acy: 0.8832 - val_loss: 0.3494 - val_accuracy: 0.8545
Epoch 45/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2465 - accur
acy: 0.8837 - val_loss: 0.3739 - val_accuracy: 0.8648
Epoch 46/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2514 - accur
acy: 0.8833 - val_loss: 0.3594 - val_accuracy: 0.8689
Epoch 47/100
23/23 [==============================] - 0s 22ms/step - loss: 0.2447 - accur
acy: 0.8836 - val_loss: 0.3539 - val_accuracy: 0.8648
Epoch 48/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2423 - accur
acy: 0.8863 - val_loss: 0.3625 - val_accuracy: 0.8525
Epoch 49/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2432 - accur
acy: 0.8869 - val_loss: 0.3554 - val_accuracy: 0.8566
Epoch 50/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2404 - accur
acy: 0.8860 - val_loss: 0.3577 - val_accuracy: 0.8586
Epoch 51/100
23/23 [==============================] - 1s 24ms/step - loss: 0.2410 - accur
acy: 0.8858 - val_loss: 0.3585 - val_accuracy: 0.8525
Epoch 52/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2397 - accur
acy: 0.8866 - val_loss: 0.3752 - val_accuracy: 0.8463
Epoch 53/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2374 - accur
acy: 0.8885 - val_loss: 0.3613 - val_accuracy: 0.8607
Epoch 54/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2359 - accur
acy: 0.8877 - val_loss: 0.3611 - val_accuracy: 0.8607
Epoch 55/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2415 - accur
acy: 0.8870 - val_loss: 0.3674 - val_accuracy: 0.8627
Epoch 56/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2381 - accur
acy: 0.8884 - val_loss: 0.3754 - val_accuracy: 0.8443
```

```
Epoch 57/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2351 - accur
acy: 0.8898 - val_loss: 0.3750 - val_accuracy: 0.8545
Epoch 58/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2362 - accur
acy: 0.8886 - val_loss: 0.3584 - val_accuracy: 0.8689
Epoch 59/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2338 - accur
acy: 0.8871 - val_loss: 0.3633 - val_accuracy: 0.8545
Epoch 60/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2367 - accur
acy: 0.8881 - val_loss: 0.3700 - val_accuracy: 0.8607
Epoch 61/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2372 - accur
acy: 0.8886 - val_loss: 0.3648 - val_accuracy: 0.8586
Epoch 62/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2361 - accur
acy: 0.8870 - val_loss: 0.3596 - val_accuracy: 0.8422
Epoch 63/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2378 - accur
acy: 0.8886 - val_loss: 0.3546 - val_accuracy: 0.8525
Epoch 64/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2340 - accur
acy: 0.8902 - val_loss: 0.3504 - val_accuracy: 0.8648
Epoch 65/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2316 - accur
acy: 0.8906 - val_loss: 0.3530 - val_accuracy: 0.8586
Epoch 66/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2343 - accur
acy: 0.8877 - val_loss: 0.3518 - val_accuracy: 0.8525
Epoch 67/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2368 - accur
acy: 0.8883 - val_loss: 0.3570 - val_accuracy: 0.8607
Epoch 68/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2314 - accur
acy: 0.8893 - val_loss: 0.3680 - val_accuracy: 0.8525
Epoch 69/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2325 - accur
acy: 0.8887 - val_loss: 0.3487 - val_accuracy: 0.8586
Epoch 70/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2320 - accur
acy: 0.8925 - val_loss: 0.3654 - val_accuracy: 0.8566
Epoch 71/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2320 - accur
acy: 0.8922 - val_loss: 0.3515 - val_accuracy: 0.8586
Epoch 72/100
23/23 [==============================] - 0s 19ms/step - loss: 0.2291 - accur
acy: 0.8949 - val_loss: 0.3566 - val_accuracy: 0.8586
Epoch 73/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2373 - accur
acy: 0.8895 - val_loss: 0.3567 - val_accuracy: 0.8750
Epoch 74/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2353 - accur
acy: 0.8894 - val_loss: 0.3384 - val_accuracy: 0.8750
Epoch 75/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2325 - accur
```

```
acy: 0.8891 - val_loss: 0.3475 - val_accuracy: 0.8730
Epoch 76/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2380 - accur
acy: 0.8877 - val_loss: 0.3623 - val_accuracy: 0.8586
Epoch 77/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2343 - accur
acy: 0.8900 - val_loss: 0.3536 - val_accuracy: 0.8750
Epoch 78/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2297 - accur
acy: 0.8911 - val_loss: 0.3525 - val_accuracy: 0.8607
Epoch 79/100
23/23 [==============================] - 0s 22ms/step - loss: 0.2295 - accur
acy: 0.8912 - val_loss: 0.3450 - val_accuracy: 0.8750
Epoch 80/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2270 - accur
acy: 0.8940 - val_loss: 0.3513 - val_accuracy: 0.8586
Epoch 81/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2282 - accur
acy: 0.8913 - val_loss: 0.3474 - val_accuracy: 0.8607
Epoch 82/100
23/23 [==============================] - 1s 26ms/step - loss: 0.2281 - accur
acy: 0.8913 - val_loss: 0.3598 - val_accuracy: 0.8627
Epoch 83/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2333 - accur
acy: 0.8905 - val_loss: 0.3557 - val_accuracy: 0.8709
Epoch 84/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2323 - accur
acy: 0.8897 - val_loss: 0.3537 - val_accuracy: 0.8709
Epoch 85/100
23/23 [==============================] - 1s 25ms/step - loss: 0.2314 - accur
acy: 0.8915 - val_loss: 0.3460 - val_accuracy: 0.8627
Epoch 86/100
23/23 [==============================] - 0s 22ms/step - loss: 0.2305 - accur
acy: 0.8921 - val_loss: 0.3519 - val_accuracy: 0.8586
Epoch 87/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2307 - accur
acy: 0.8926 - val_loss: 0.3529 - val_accuracy: 0.8709
Epoch 88/100
23/23 [==============================] - 1s 22ms/step - loss: 0.2294 - accur
acy: 0.8934 - val_loss: 0.3530 - val_accuracy: 0.8689
Epoch 89/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2338 - accur
acy: 0.8885 - val_loss: 0.3545 - val_accuracy: 0.8668
Epoch 90/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2316 - accur
acy: 0.8918 - val_loss: 0.3472 - val_accuracy: 0.8627
Epoch 91/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2309 - accur
acy: 0.8904 - val_loss: 0.3605 - val_accuracy: 0.8545
Epoch 92/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2318 - accur
acy: 0.8898 - val_loss: 0.3721 - val_accuracy: 0.8627
Epoch 93/100
23/23 [==============================] - 0s 19ms/step - loss: 0.2332 - accur
acy: 0.8921 - val_loss: 0.3446 - val_accuracy: 0.8709
Epoch 94/100
```

```
23/23 [==============================] - 1s 22ms/step - loss: 0.2318 - accur
acy: 0.8919 - val_loss: 0.3574 - val_accuracy: 0.8648
Epoch 95/100
23/23 [==============================] - 0s 21ms/step - loss: 0.2292 - accur
acy: 0.8935 - val_loss: 0.3698 - val_accuracy: 0.8730
Epoch 96/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2338 - accur
acy: 0.8885 - val_loss: 0.3414 - val_accuracy: 0.8709
Epoch 97/100
23/23 [==============================] - 0s 22ms/step - loss: 0.2337 - accur
acy: 0.8927 - val_loss: 0.3423 - val_accuracy: 0.8627
Epoch 98/100
23/23 [==============================] - 0s 19ms/step - loss: 0.2319 - accur
acy: 0.8927 - val_loss: 0.3421 - val_accuracy: 0.8668
Epoch 99/100
23/23 [==============================] - 0s 20ms/step - loss: 0.2326 - accur
acy: 0.8895 - val_loss: 0.3637 - val_accuracy: 0.8525
Epoch 100/100
23/23 [==============================] - 1s 23ms/step - loss: 0.2272 - accur
acy: 0.8945 - val_loss: 0.3658 - val_accuracy: 0.8566
```

To avoid training and retraining the model, I'll now save this model with these results.

In [ ]:
```
nep_eng_translator.save('models/nep-eng-translator.keras')
```

Now, for future testing, I can reopen the model.

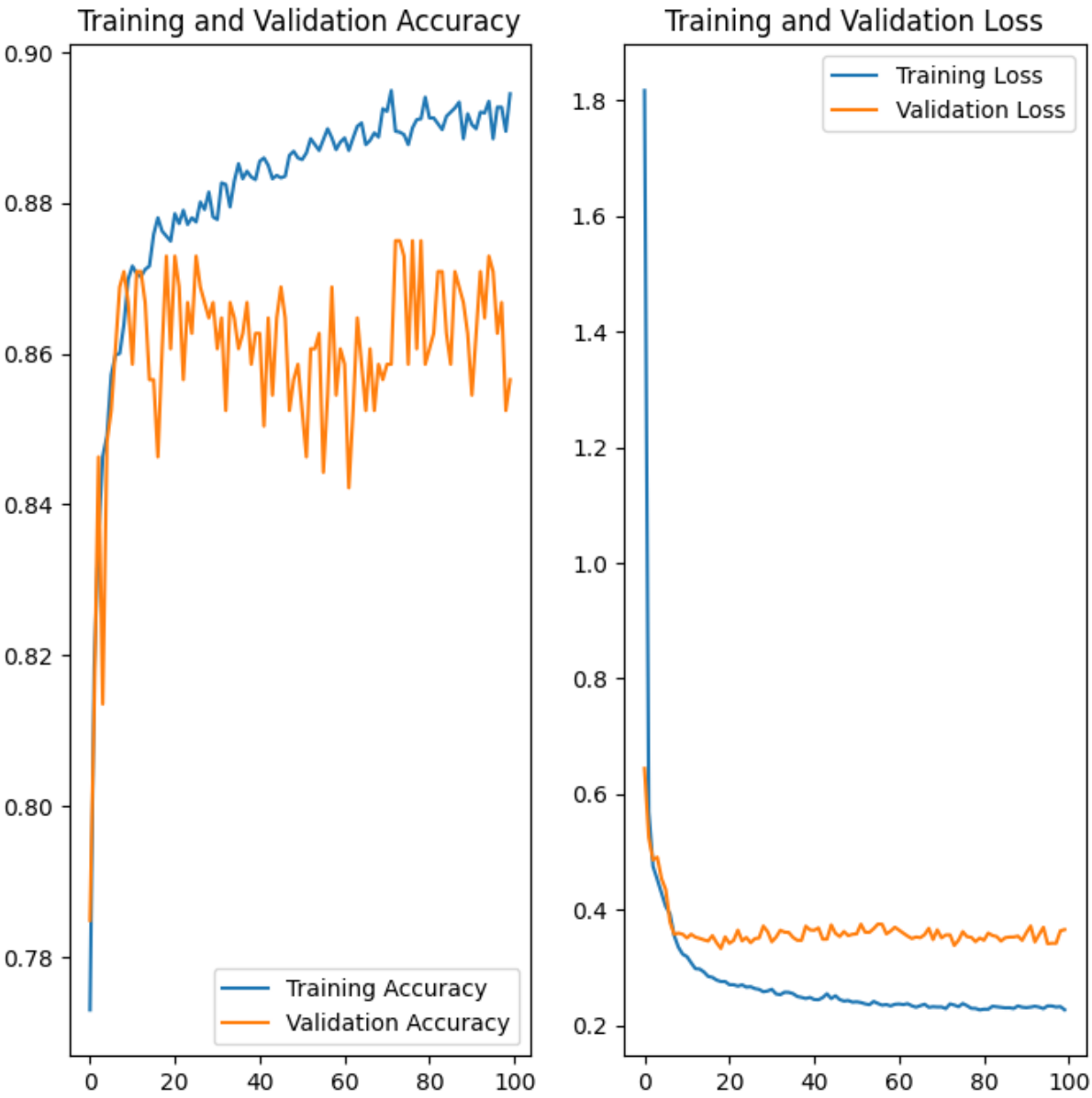
In [ ]:
```
nep_eng_translator = tf.keras.models.load_model('models/nep-eng-translator.k
```

Visualizing the Training Results

Below we can see a visualization of how loss and accuracy evolved over the course of training. Surprisingly, performance is substantially better in this model!

In [ ]:
```
acc = nep_eng_history.history['accuracy']
val_acc = nep_eng_history.history['val_accuracy']

loss = nep_eng_history.history['loss']
val_loss = nep_eng_history.history['val_loss']

epochs_range = range(100)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
```

```
plt.title('Training and Validation Loss')
plt.show()
```



English Language Text Generator

For the text generator, I will create a mini-GPT model for text generation uses the KerasNLP library. This model will be trained on the simplebooks-92 dataset. This dataset uses a simplified English vocabulary. This is useful both for training purposes and for creating generated output that is readily understandable by individuals who do not speak English as a first language.

To create this model, I drew on the Keras suggested tutorial for text generation that can be found here:

https://keras.io/examples/generative/text_generation_gpt/

Settings

Below I've selected some key hyperparameters. Particularly notable here is the minimum traing sequence length. This sets the smallest number of tokens that will be examined by the model during training. We want this number to be large enough that the model is not attempting to train on single words or brief phrases which may eat up training time while providing little in the way of performance improvements.

Additionally, NUM_TOKENS_TO_GENERATE decides how many tokens will appear in the output of the model. I've set this value to be relatively low. There are two reasons for this.

First, in testing, shorter outputs were more likely to be coherent sentences that expressed a single proposition. Given that outputs are meant to be translated and understood, coherence is extremely important.

Second, because the output sentences are meant to be translated and learned by a student, we want them to be short and simple enough to be readily understood and learned, even by a beginner.

In [ ]:
```
BATCH_SIZE = 64
SEQ_LEN = 128
MIN_TRAINING_SEQ_LEN = 450

EMBED_DIM = 256
FEED_FORWARD_DIM = 256
NUM_HEADS = 3
NUM_LAYERS = 2
VOCAB_SIZE = 5000

EPOCHS = 50

NUM_CHAR_TO_GENERATE = 50
```

Load Simplebooks Data

In [ ]:
```
raw_train_ds = (
    tf.data.TextLineDataset('datasets/simplebooks/simplebooks-92-raw/train.t
    .filter(lambda x: tf.strings.length(x) > MIN_TRAINING_SEQ_LEN)
    .batch(BATCH_SIZE)
    .shuffle(buffer_size=256)
)

raw_val_ds = (
    tf.data.TextLineDataset("datasets/simplebooks/simplebooks-92-raw/valid.t
    .filter(lambda x: tf.strings.length(x) > MIN_TRAINING_SEQ_LEN)
    .batch(BATCH_SIZE)
)
```

Tokenizer for Generator

Here I've defined the vocabulary for the model. This vocabulary is made up of words ('tokens') from the dataset that the model needs to be able to represent and understand.

PAD, UNK, BOS represent padding, unknown, and beginning-of-sentence. These tokens are set aside as non-words for our purposes.

I've then loaded in KerasNLP's tokenizer and used it to preprocess the data for training. This strips out punctuation, sets every word to be all lowercase and then assigns a unique integer to each word. This allows the model to train on the data as quantified data.

Although I've already created a vocabulary and tokenizer for the translator models above, because the set of English sentences in the translation data set is so small, it is necessary to create new ones for the generator model.

Note, if you are re-running this notebook, the vocabulary computation is quite slow.

In [ ]:
```
vocab = keras_nlp.tokenizers.compute_word_piece_vocabulary(
    raw_train_ds,
    vocabulary_size = VOCAB_SIZE,
    lowercase = True,
    reserved_tokens = ["[PAD]", "[UNK]", "[BOS]"],
)
```

In [ ]:
```
gen_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary=vocab,
    sequence_length=SEQ_LEN,
    lowercase=True,
)
```

In [ ]:
```
start_packer = keras_nlp.layers.StartEndPacker(
    sequence_length=SEQ_LEN,
    start_value=gen_tokenizer.token_to_id("[BOS]")
)

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

train_ds = raw_train_ds.map(tf.autograph.experimental.do_not_convert(preproc

val_ds = raw_val_ds.map(tf.autograph.experimental.do_not_convert(preprocess)
```

Constructing the Model

Now we can actually create the model. This model is another Auto-encoder.

This model is very similar architecturally to the translator model. Tokens are passed to an input layer and then to an embedding layer which accounts for both the number

associated with the token as well as the position of the token in its sentential context. This then downsamples the sentence to an embedding.

These embeddings are then passed to a decoder layer that attempts to generate a sentence by upsampling from the embedding.

In [ ]:
```
inputs = keras.layers.Input(shape=(None,), dtype=tf.int32)

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)

for _ in range(NUM_LAYERS):
    decoder_layer = keras_nlp.layers.TransformerDecoder(
        num_heads=NUM_HEADS,
        intermediate_dim=FEED_FORWARD_DIM,
    )
    x = decoder_layer(x)

outputs = keras.layers.Dense(VOCAB_SIZE)(x)
generator = keras.Model(inputs=inputs, outputs=outputs)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Below, I've compiled the model. As before, the loss function is Sparse Categorical Crossentropy. Again, natural language is almost always going to be sparse data so this is the natural choice.

Here I've used Adam as the optimizer. The smallness of the dataset isn't really a concern here so I've chosen Adam for it's ability to produce higher accuracy, even though the computation costs are a little higher.

In [ ]:
```
generator.compile(optimizer='adam', loss=loss_fn, metrics=[])
```

Model Summary

In [ ]:
```
generator.summary()
```

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, None)]            0

 token_and_position_embeddi  (None, None, 256)         1312768
 ng_3 (TokenAndPositionEmbe
 dding)

 transformer_decoder_3 (Tra  (None, None, 256)         394749
 nsformerDecoder)

 transformer_decoder_4 (Tra  (None, None, 256)         394749
 nsformerDecoder)

 dense_16 (Dense)            (None, None, 5000)        1285000

=================================================================
Total params: 3387266 (12.92 MB)
Trainable params: 3387266 (12.92 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Model Training

Below, I've trained the model. Note that loss is dropping very slowly, which is to say, for a dataset this size the model converges very slowly. For a final product, far more epochs would be needed to achieve satisfactory results.

In [ ]:
```
generator.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
```

```
Epoch 1/50
   3168/Unknown - 126s 38ms/step - loss: 3.7765
2023-08-18 11:59:29.606552: I tensorflow/core/framework/local_rendezvous.cc:
409] Local rendezvous send item cancelled. Key hash: 6603943071277581536
2023-08-18 11:59:29.606607: I tensorflow/core/framework/local_rendezvous.cc:
409] Local rendezvous send item cancelled. Key hash: 4098329605364189913
2023-08-18 11:59:29.606640: I tensorflow/core/framework/local_rendezvous.cc:
409] Local rendezvous send item cancelled. Key hash: 16341194706631034183
3169/3169 [==============================] - 127s 38ms/step - loss: 3.7765 -
val_loss: 3.8371
2023-08-18 11:59:30.314296: I tensorflow/core/framework/local_rendezvous.cc:
409] Local rendezvous send item cancelled. Key hash: 8472384756783010550
2023-08-18 11:59:30.314341: I tensorflow/core/framework/local_rendezvous.cc:
409] Local rendezvous send item cancelled. Key hash: 1432970336118965353
```

```
Epoch 2/50
3169/3169 [==============================] - 121s 37ms/step - loss: 3.7560 -
val_loss: 3.7982
Epoch 3/50
3169/3169 [==============================] - 110s 33ms/step - loss: 3.7386 -
val_loss: 3.7830
Epoch 4/50
3169/3169 [==============================] - 107s 33ms/step - loss: 3.7217 -
val_loss: 3.8098
Epoch 5/50
3169/3169 [==============================] - 119s 36ms/step - loss: 3.7087 -
val_loss: 3.7441
Epoch 6/50
3169/3169 [==============================] - 110s 32ms/step - loss: 3.6967 -
val_loss: 3.7557
Epoch 7/50
3169/3169 [==============================] - 102s 31ms/step - loss: 3.6859 -
val_loss: 3.7628
Epoch 8/50
3169/3169 [==============================] - 101s 31ms/step - loss: 3.6766 -
val_loss: 3.7362
Epoch 9/50
3169/3169 [==============================] - 106s 32ms/step - loss: 3.6688 -
val_loss: 3.7375
Epoch 10/50
3169/3169 [==============================] - 108s 33ms/step - loss: 3.6617 -
val_loss: 3.7417
Epoch 11/50
3169/3169 [==============================] - 130s 39ms/step - loss: 3.6559 -
val_loss: 3.7191
Epoch 12/50
3169/3169 [==============================] - 114s 35ms/step - loss: 3.6492 -
val_loss: 3.7226
Epoch 13/50
3169/3169 [==============================] - 119s 36ms/step - loss: 3.6436 -
val_loss: 3.6952
Epoch 14/50
3169/3169 [==============================] - 116s 35ms/step - loss: 3.6389 -
val_loss: 3.6960
Epoch 15/50
3169/3169 [==============================] - 117s 35ms/step - loss: 3.6338 -
val_loss: 3.7120
Epoch 16/50
3169/3169 [==============================] - 111s 34ms/step - loss: 3.6303 -
val_loss: 3.6950
Epoch 17/50
3169/3169 [==============================] - 109s 33ms/step - loss: 3.6262 -
val_loss: 3.7143
Epoch 18/50
3169/3169 [==============================] - 109s 33ms/step - loss: 3.6224 -
val_loss: 3.7224
Epoch 19/50
3169/3169 [==============================] - 112s 34ms/step - loss: 3.6198 -
val_loss: 3.7073
Epoch 20/50
3169/3169 [==============================] - 113s 34ms/step - loss: 3.6160 -
```

```
val_loss: 3.7016
Epoch 21/50
3169/3169 [==============================] - 130s 40ms/step - loss: 3.6131 -
val_loss: 3.6911
Epoch 22/50
3169/3169 [==============================] - 121s 37ms/step - loss: 3.6101 -
val_loss: 3.7070
Epoch 23/50
3169/3169 [==============================] - 140s 41ms/step - loss: 3.6079 -
val_loss: 3.6862
Epoch 24/50
3169/3169 [==============================] - 133s 40ms/step - loss: 3.6052 -
val_loss: 3.6918
Epoch 25/50
3169/3169 [==============================] - 132s 40ms/step - loss: 3.6027 -
val_loss: 3.6840
Epoch 26/50
3169/3169 [==============================] - 124s 38ms/step - loss: 3.6007 -
val_loss: 3.6916
Epoch 27/50
3169/3169 [==============================] - 135s 40ms/step - loss: 3.5995 -
val_loss: 3.6916
Epoch 28/50
3169/3169 [==============================] - 121s 36ms/step - loss: 3.5968 -
val_loss: 3.7013
Epoch 29/50
3169/3169 [==============================] - 125s 38ms/step - loss: 3.5948 -
val_loss: 3.6827
Epoch 30/50
3169/3169 [==============================] - 115s 35ms/step - loss: 3.5935 -
val_loss: 3.6793
Epoch 31/50
3169/3169 [==============================] - 117s 35ms/step - loss: 3.5916 -
val_loss: 3.6709
Epoch 32/50
3169/3169 [==============================] - 126s 38ms/step - loss: 3.5893 -
val_loss: 3.6897
Epoch 33/50
3169/3169 [==============================] - 129s 39ms/step - loss: 3.5879 -
val_loss: 3.7001
Epoch 34/50
3169/3169 [==============================] - 116s 35ms/step - loss: 3.5865 -
val_loss: 3.6823
Epoch 35/50
3169/3169 [==============================] - 112s 34ms/step - loss: 3.5849 -
val_loss: 3.6880
Epoch 36/50
3169/3169 [==============================] - 112s 34ms/step - loss: 3.5830 -
val_loss: 3.6953
Epoch 37/50
3169/3169 [==============================] - 111s 34ms/step - loss: 3.5816 -
val_loss: 3.6737
Epoch 38/50
3169/3169 [==============================] - 113s 34ms/step - loss: 3.5805 -
val_loss: 3.6803
Epoch 39/50
```

```
3169/3169 [==============================] - 108s 33ms/step - loss: 3.5789 -
val_loss: 3.6807
Epoch 40/50
3169/3169 [==============================] - 106s 32ms/step - loss: 3.5783 -
val_loss: 3.6953
Epoch 41/50
3169/3169 [==============================] - 120s 37ms/step - loss: 3.5761 -
val_loss: 3.6894
Epoch 42/50
3169/3169 [==============================] - 113s 34ms/step - loss: 3.5759 -
val_loss: 3.6887
Epoch 43/50
3169/3169 [==============================] - 107s 33ms/step - loss: 3.5748 -
val_loss: 3.6860
Epoch 44/50
3169/3169 [==============================] - 117s 36ms/step - loss: 3.5730 -
val_loss: 3.6890
Epoch 45/50
3169/3169 [==============================] - 110s 33ms/step - loss: 3.5726 -
val_loss: 3.6725
Epoch 46/50
3169/3169 [==============================] - 114s 35ms/step - loss: 3.5713 -
val_loss: 3.6919
Epoch 47/50
3169/3169 [==============================] - 108s 33ms/step - loss: 3.5709 -
val_loss: 3.6709
Epoch 48/50
3169/3169 [==============================] - 110s 33ms/step - loss: 3.5694 -
val_loss: 3.6882
Epoch 49/50
3169/3169 [==============================] - 111s 34ms/step - loss: 3.5683 -
val_loss: 3.6811
Epoch 50/50
3169/3169 [==============================] - 109s 33ms/step - loss: 3.5672 -
val_loss: 3.6706
```

Out[ ]: <keras.src.callbacks.History at 0x7f908d113a00>

Like before, I'll save and reload the model for future testing.

In [ ]:
```
# generator.save('models/text-generator.keras')
```

In [ ]:
```
generator = tf.keras.models.load_model('models/text-generator.keras')
```

Testing the Text Generator

Let's see what sort of results we're getting from the generator.

Like for the translator models, we need to select a sampler. The sampler is the algorithm used to pick the next work in our output sequence based on the probabilities that were calculated by the model.

There are a a number of samplers to choose from. I've found the best results, the most

coherent sentences, from using the Top P Sampler.

The Top P Sampler takes a probability, say .9, and selects the most likely sequence of words that sum to that probability. This is different from other samplers which only select the most probable next single word. By using Top P we can get outputs that more closely resemble coherent phrases rather than strings of words that only make sense in context of the couple of words surrounding them.

Below, I've defined a "generate" function which takes in a prompt and returns a sentence generated by our model. Note that the generated outputs need to be cleaned before they are presentable. Like with the translations they are stripped of tokens that simply convey the beginnings and endings of sentences as well as unnecessary whitespace and potentially confusing punctuation.

In [ ]:
```
def generate():

    def next(prompt, cache, index):
        logits = generator(prompt)[:, index - 1, :]
        hidden_states = None
        return logits, hidden_states, cache

    def clean(txt):
        clean_txt = txt.numpy()[0].decode("utf-8")
        clean_txt = (
        clean_txt.replace("[PAD]", "")
        .replace("[START]", "")
        .replace("[END]", "")
        .replace("[BOS]", "")
        .replace(",", "")
        .replace("!", ".")
        .replace("?", ".")
        .replace("\'", "")
        .replace("\"", "")
        .replace(";", "")
        .replace(":", "")
        )
        clean_txt = clean_txt[:NUM_CHAR_TO_GENERATE].split(".")[0].strip()

        return clean_txt

    prompt = start_packer(gen_tokenizer(['']))
    sampler = keras_nlp.samplers.TopPSampler(p=0.9)
    output_tokens = sampler(
        next=next,
        prompt = prompt,
        index=1,
    )
    txt = clean(gen_tokenizer.detokenize(output_tokens))

    return txt
```

Ok, now let's pass a prompt to the model and see what we get!

In [ ]:
```
txt = generate()

print(f"Generated text: \n{txt}\n")
```

```
Generated text:
the bulk of a wagon had departed in front of his
```

Conclusion

We now have functions that translate from English to Nepali and from Nepali to English. We also have the ability to generate novel sentences in English. Let's take a look again at some examples.

Example Translations

In [ ]:
```
test_eng_texts = [pair[0] for pair in test_pairs]
for i in range(5):
    input_sentence = random.choice(test_eng_texts)
    translated = eng_nep_translate(tf.constant([input_sentence]))
    translated = translated.numpy()[0].decode("utf-8")
    translated = (
        translated.replace("[PAD]", "")
        .replace("[START]", "")
        .replace("[END]", "")
        .strip()
    )

    print(f"** Example {i} **")
    print(input_sentence)
    print(translated)
    print()
```

```
** Example 0 **
no one brought us anything.
तिमिसँग कुनैख्ने छ ।

** Example 1 **
i didn't want to go, but i did.
मलाई मौले सक्षध्नेछु ।

** Example 2 **
sharkskin can be used like the skin of other animals.
सुनकको लागि अपष्ट वाद्डा हो ।

** Example 3 **
tom just wanted to be helpful.
टम जसरी चाहन्छु ।

** Example 4 **
this tea is too bitter.
यो अक्बि धेरै राउँदै छ ।
```

Text Generation Examples

In [ ]:
```
for i in range(5):
    txt = generate()
    print(f"Generated text {i}: \n{txt}\n")
```

```
Generated text 0:
of course he didn  t want to come out

Generated text 1:
rhoda and  on the way  the rest of the toys still

Generated text 2:
i got into the room where i had taken her in my

Generated text 3:
it is the story of the memmaest  whom you really

Generated text 4:
look
```

Translating Generated Text

Now let's look at translations of generated text.

In [ ]:
```
for i in range(5):
    input_sentence = generate()
    translated = eng_nep_translate(tf.constant([input_sentence]))
    translated = translated.numpy()[0].decode("utf-8")
    translated = (
        translated.replace("[PAD]", "")
        .replace("[START]", "")
```

```
        .replace("[END]", "")
        .strip()
    )

    print(f"** Translation of Generated Text {i} **")
    print(input_sentence)
    print(translated)
    print()
```

```
** Translation of Generated Text 0 **
as time went on their stairs  they made up their
जर्म्रो विदाको राजधानी रोद्याम्दिन मन पर्छ ।

** Translation of Generated Text 1 **
i wish it had to turn to the assault of trodden
मैले के यो शंकाउनुझ्नुभगी ल्याप्त पीचको उठाबीमा हुनुचा घाइते भए ।

** Translation of Generated Text 2 **
several men were very anxious to know all the fac
मलाई स्पष्ट रूपष्ट प्राकुक आगे केही फावना छैन ।

** Translation of Generated Text 3 **
his majesty spoke of his angry words  and was rea
आफ्नो गर्मीमा धेरै रोधैं रात्तिताई दरस्ताता त मारिए वा घाइते भए ।

** Translation of Generated Text 4 **
by daybreak  sir john began to webbergs on the mo
ब्रसेल पुचाउको नजिम्क पवित्रा चाहन्छु ।
```

So we've done it! This is a great starting place for an AI language tutor. We can translate a into and out of a new language and generate novel sentences for translation and study.

Obviously these models are only a starting point. For a more robust product we will need much more robust data and significantly more training time. This is unfortunately a problem that plagues work on global minority languages. But, hopefully with tools like these we can start to move in the right direction!