Text Translation Using Sequence to Sequence Neural Networks

In our last two chapters, we used neural networks to classify text and perform sentiment analysis. Both of these tasks involve taking an NLP input and predicting some value. In the case of our sentiment analysis, this was a number between 0 and 1 representing the sentiment of our sentence and in the case of our sentence classification model, our output was a multi-class prediction of which of several categories our sentence belonged to. But what if we wish to make not just a single prediction, but predict a whole sentence? In this chapter we will build a sequence to sequence model which takes a sentence in one language as its input and outputs the translation of this sentence in another language.

We have already explored several types of neural network architecture used for NLP learning, in previous chapters. In this chapter, we will again be using the familiar RNNs, but instead of just building a simple RNN model, we will use RNNs as part of a larger, more complex model in order to perform sequence to sequence translation. By using the underpinnings of RNNs that we have learned in the previous chapters, we can show how these concepts can be extended in order to create a variety of models that can be fit for purpose.

In this chapter we will cover the following topics:

* Theory of Sequence to Sequence Models
* Building a Sequence to Sequence Neural Network for Text Translation

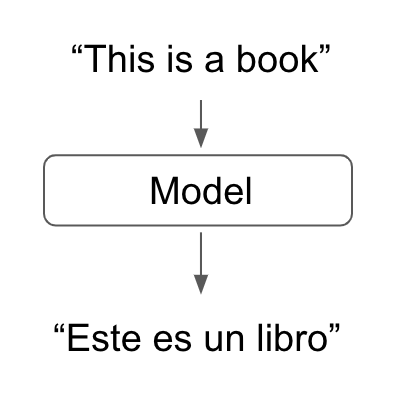
**Technical requirements**

All code for this chapter can be found at : https://github.com/PacktPublishing/Hands-On-Natural-Language-Processing-with-PyTorch-1.x.

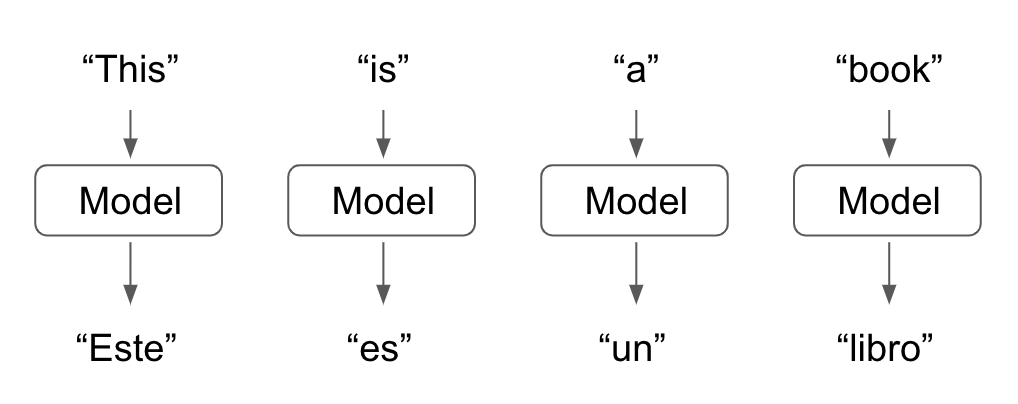
**Theory of Sequence to Sequence Models**

Sequence to sequence models are very similar to more conventional neural network structures we have seen so far. The main difference being that for a model’s output, we expect another sequence, rather than a binary or multi-class prediction. This is particularly useful in tasks such as translation where we may wish to convert a whole sentence to another language.

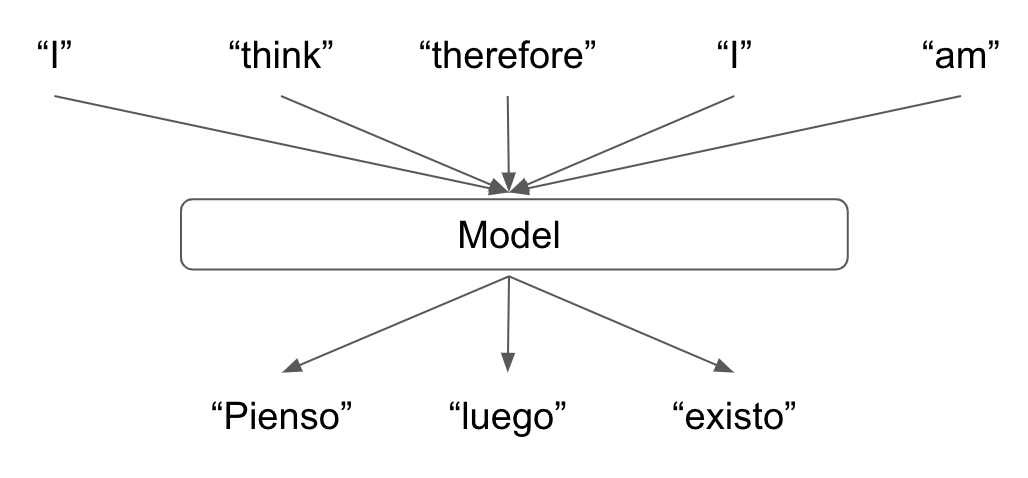
In the following example, we see that our English to Spanish translation maps word to word:



The first word in our input sentence maps nicely to the first word in our output sentence. If this were the case for all languages, we could simply pass each word in our sentence one by one through our trained model to get an output sentence and there would be no need for any sequence to sequence modelling, as shown here:



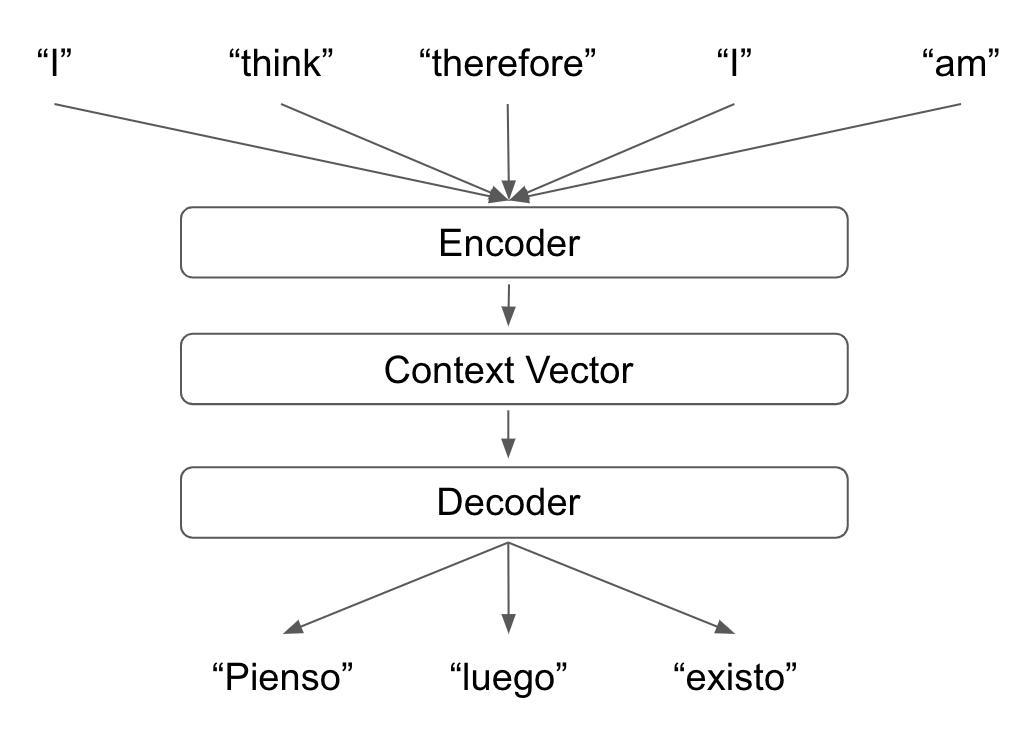
However, we know from our experience with NLP that language is not as simple as this! Single words in one language may map to multiple words in other languages and the other in which these words occur in a grammatically correct sentence may not be the same. Therefore, we need a model that is capable of capturing the context of a whole sentence and outputting a correct translation, not a model that aims to directly translate individual words. This is where sequence to sequence modelling becomes essential, as seen here:



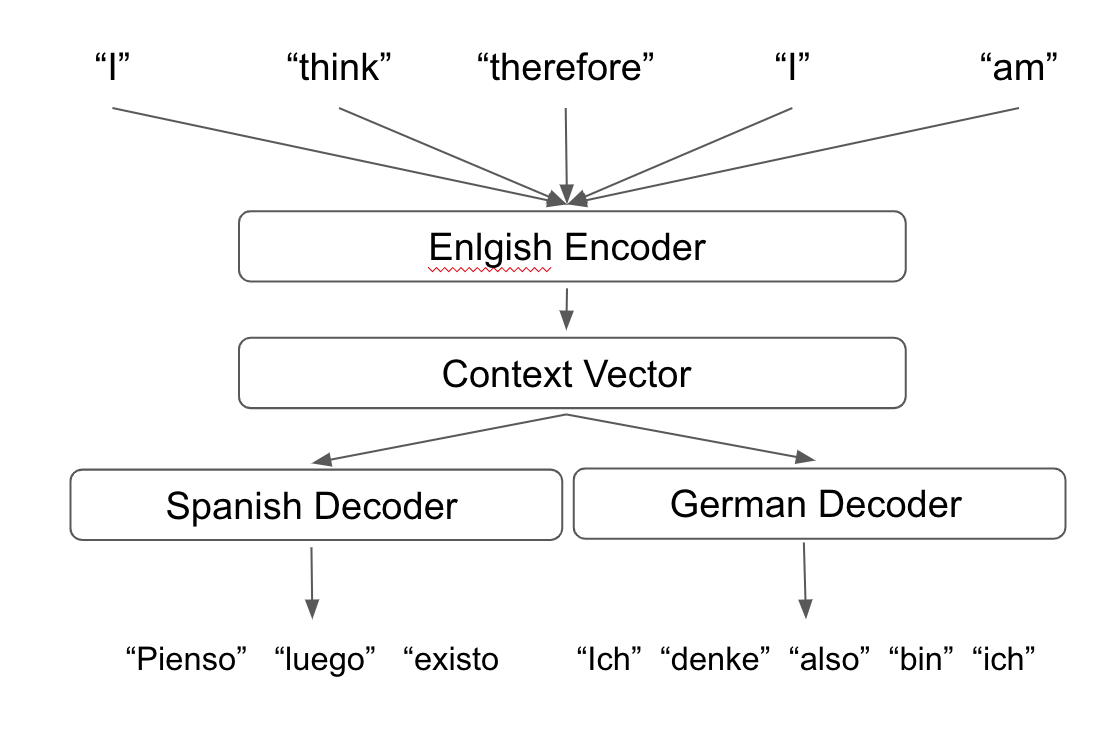
To train a sequence to sequence model that captures the context of the input sentence and translates this into an output sentence, we will essentially train two smaller models that allow us to do this:

* An **encoder** model which captures the context of our sentence and outputs it as a single context vector.
* A **decoder** which takes the context vector representation of our original sentence and translates this into a different language.

So in reality, our full sequence to sequence translation model will actually look something like this:



By splitting our models into individual encoder and decoder elements, we are effectively modularizing our models. This means that if we wish to train multiple models to translate from English into different languages, we do not need to retrain the whole model each time. We need to only train multiple different decoders to transform our context vector into our output sentences. Then when making predictions, we can simply substitute in the decoder that we wish to use for our translation:

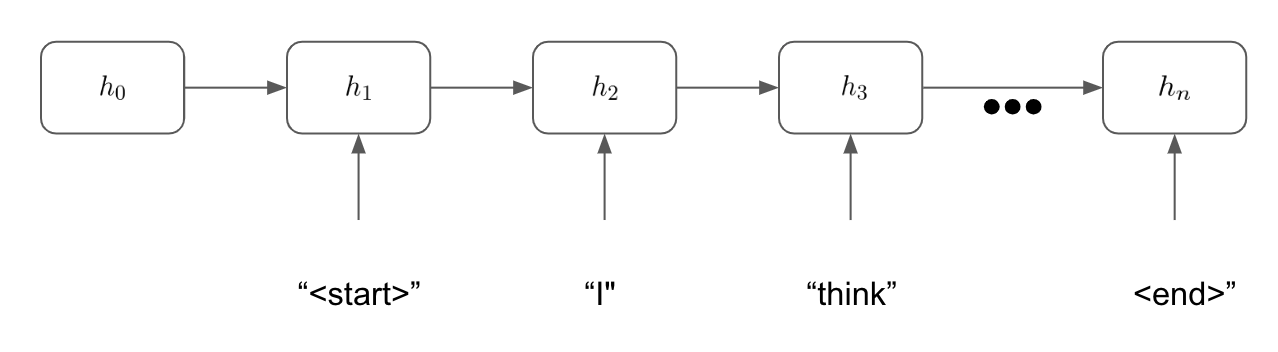


Next, we will examine the encoder and decoder components of the sequence to sequence model individually.

**Encoder**

The purpose of the encoder element of our sequence to sequence model is to be able to fully capture the context of our input sentence and represent it as a vector. We can do this by using RNNs or more specifically, LSTMs. As we recall from our previous chapters, RNNs take a sequential input and maintain a hidden state throughout this sequence. Each new word in the sequence updates the hidden state and at the end of the sequence we can use the model’s final hidden state as our input into our next layer.

In the case of our encoder, the hidden state represents the context vector representation of our whole sentence, meaning we can use the hidden state output of our RNN to represent the entirety of the input sentence:

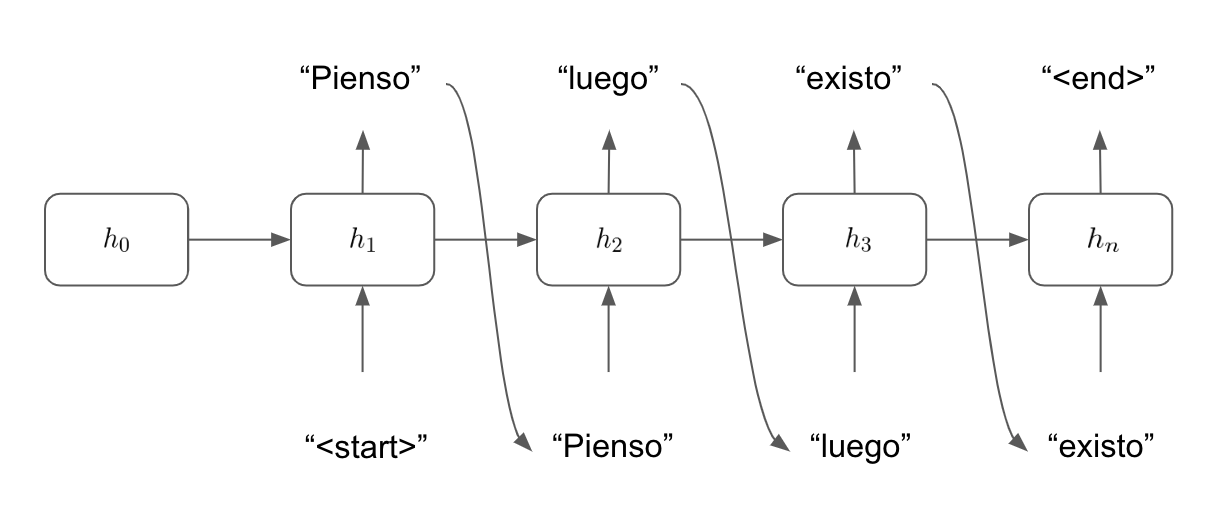


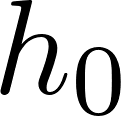
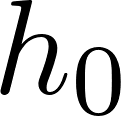
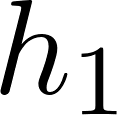
We use our final hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bn%7D%250) as our context vector which we will then decode using a trained decoder. It is also worth observing that in the context of our sequence to sequence models, we append a “start” and “end” tokens to the beginning and ends of our input sentence. This is because our inputs and outputs do not have finite length and our model needs to be able to learn when a sentence should end. Our input sentence will always end with an “end” token which signals to the encoder that the hidden state at this point will be used as the final context vector representation for this input sentence. Similarly, in the decoder step we will see that our decoder will keep generating words until it predicts an “end” token. This allows our decoder to generate actual output sentences as opposed to a sequence of tokens of infinite length.

Next we will look at how the decoder takes this context vector and learns to translate it into an output sentence.

**Decoder**

Our decoder takes the final hidden state from our encoder layer and decodes this into a sentence to another language. Our decoder is an RNN similar to that of our encoder, but while our encoder updates its hidden state given its current hidden state and the current word in the sentence, our decoder updates it’s hidden state and outputs a token at each iteration, given the current hidden state and the previous predicted word in the sentence. This can be seen in the following diagram:



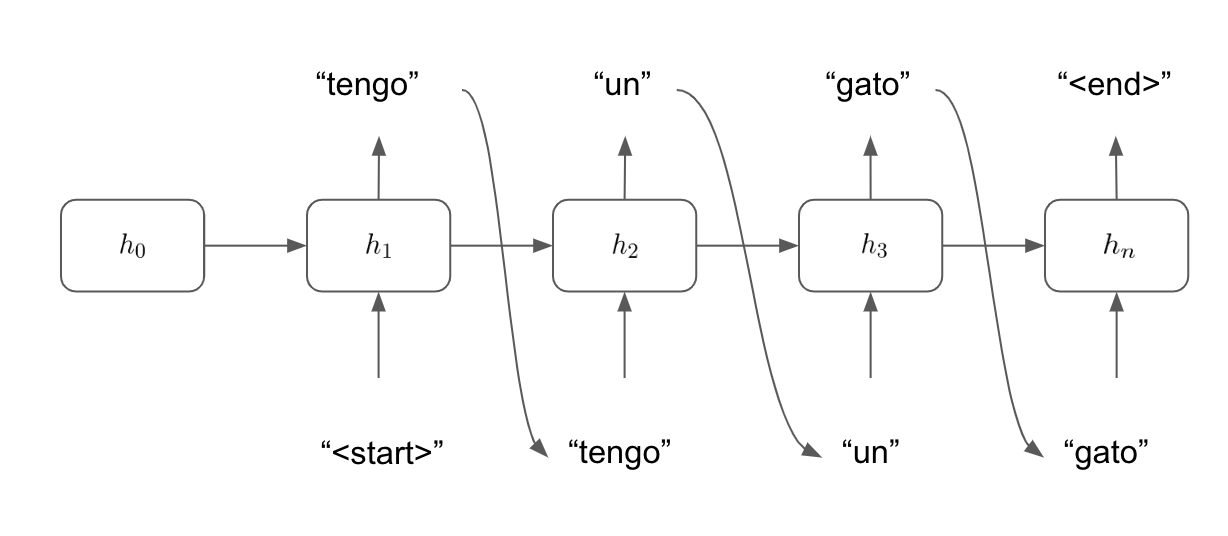
Our model first takes the context vector as the final hidden state from our encoder step [](https://www.codecogs.com/eqnedit.php?latex=h_%7B0%7D%250). Our model then aims to predict the next word in the sentence given the current hidden state and then previous word in the sentence. We know our sentence must begin with a “start” token so at our first step, our model tries to predict the first word in the sentence given the previous hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7B0%7D%250) and the previous word in the sentence (in this instance, the “start” token). Our model makes a prediction (“pienso”) and then updates the hidden state to reflect the new state of the model [](https://www.codecogs.com/eqnedit.php?latex=h_%7B1%7D%250). Then at the next step our model uses the new hidden state and the last predicted word to predict the next word in the sentence. This continues until the model predicts the “end” token, at which point our model stops generating output words.

The intuition behind this model is in line with what we have learnt about language representations thus far. Words in any given sentence are dependent on the words that come before it. So to predict any given word in a sentence without considering the words that have been predicted before, it would not make sense as words in any given sentence are not independent from one another.

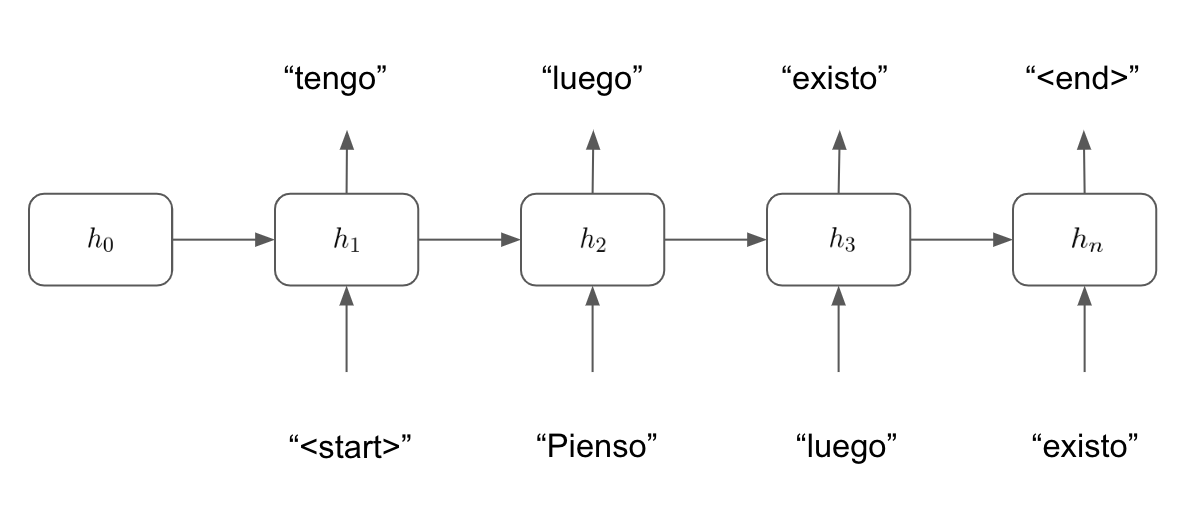
We learn our model parameters as we have done before, by making a forward pass, calculating the loss of our target sentence against the predicted sentence and backpropagating this loss through the network, updating the parameters as we go. However, learning using this process can be very slow as to begin with our model will have very little predictive power. As our predictions of words in our target sentence are not independent of one another, if we predict the first word in our target sentence incorrectly, subsequent words in our output sentence are also unlikely to be correct. To help with this process we can use a technique known as **teacher forcing**.

**Using Teacher Forcing**

As our model initially does not make good predictions, we will find that any initial errors are multiplied exponentially. If our first predicted word in the sentence is incorrect then the rest of the sentence will likely be incorrect as well. This is because the predictions our model makes are dependent on the previous predictions it makes. This means that any losses our model has can be multiplied exponentially and we may face the exploding gradient problem making it very difficult for our model to learn anything:



However, by using **teacher forcing**, we train our model using the correct previous target word so that one wrong prediction does not inhibit our model’s ability to learn from the correct model predictions. This means that if our model makes a wrong prediction at one point in the sentence, it can still make correct predictions using subsequent words. While our model will still have incorrectly predicted words and will have losses by which we can update our gradients, we now do not suffer exploding gradients and our model will learn much more quickly.



You can consider teacher forcing a way of helping our model to learn independently of its previous predictions at each time step so that the losses incurred by a mis-prediction at an early time step are not carried over to later time steps.

By combining the encoder and decoder steps and applying teacher forcing to help our model learn, we can build a sequence to sequence model that will allow us to translate sequences of one language into another. In the next section, we will illustrate how we can build this from scratch using PyTorch.

**Building a Sequence to Sequence Model for Text Translation**

In order to build our sequence to sequence model for translation we will implement the encoder/decoder framework we outlined previously, showing how the two halves of our model can be utilized together in order to capture a representation of our data using the encoder and then translate this representation to another language using our decoder. In order to do this, we first need to obtain our data.

**Preparing the data**

By now, we know enough about machine learning to know that for a task like this we will need a set of training data with corresponding labels. In this case, we will need **sentences in one language with the corresponding translations in another language.** Fortunately, the Torchtext library that we used in the previous chapter contains a dataset that will allow us to get this.

The Multi30kdataset in Torchtext consists of approximately 30,000 sentences with corresponding translations in multiple languages. For this translation task we will take our input sentences as English and our output sentences as German. Our fully trained model will therefore allow us to **translate English sentences into German.**

We start by extracting our data and preprocessing it. We will once again use spacy which contains an inbuilt dictionary of vocabularies which we can use to tokenize our data.

1. We start by loading our spacy tokenizers into Python. We will need to do this once for each language we are using as we will be building two entirely separate vocabularies for this task:

spacy\_german = spacy.load('de')

spacy\_english = spacy.load('en')

**Important note**

You may have to install the German vocabulary from the command line by doing the following (we already installed the English vocabulary in the previous chapter):

python3 -m spacy download de

1. Next, we create a function for each of our languages to tokenize our sentences. Note that our tokenizer for our input English sentence reverses the order of the tokens:

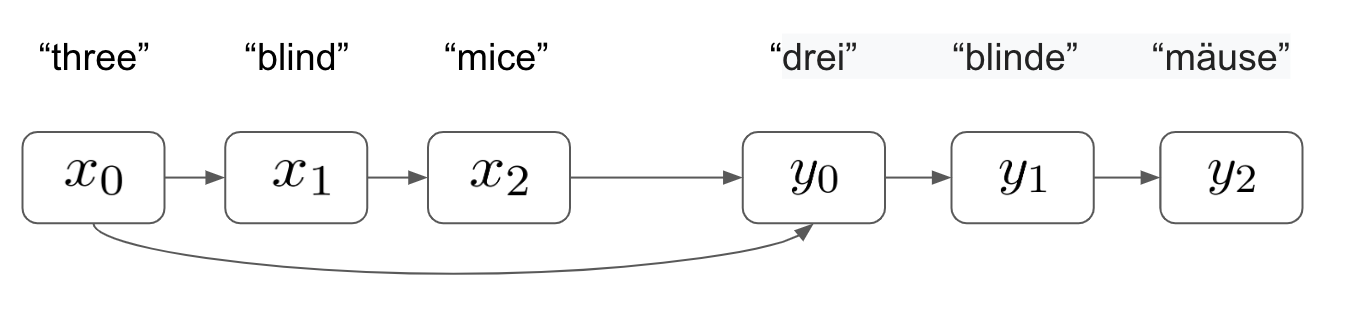
def tokenize\_german(text):

return [token.text for token in spacy\_german.tokenizer(text)]

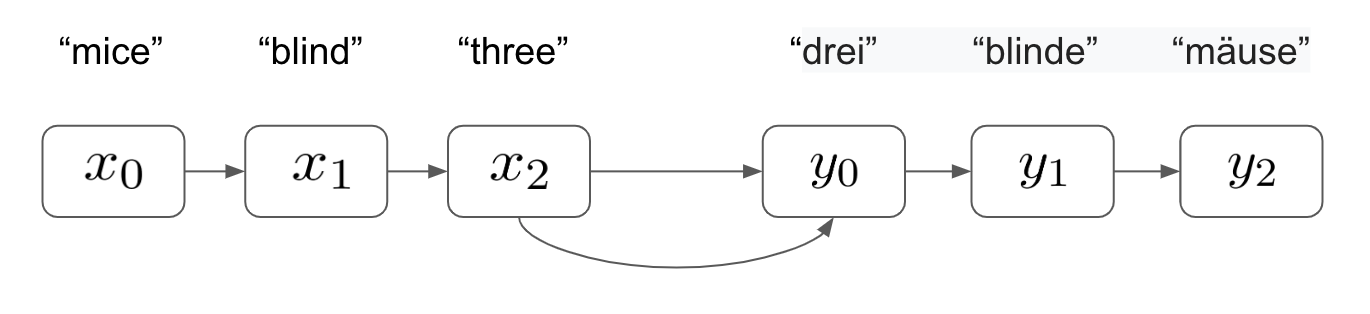
def tokenize\_english(text):

return [token.text for token in spacy\_english.tokenizer(text)][::-1]

While reversing the order of our input sentence is not compulsory, it has been shown to improve the model’s ability to learn. If our model consists of two RNNs joined together we can show that the information flow within our model is improved when reversing the input sentence. If we take a basic input sentence and English and don’t reverse it:



We see that in order to predict the first output word [](https://www.codecogs.com/eqnedit.php?latex=y_%7B0%7D%250) correctly, our first English word from [](https://www.codecogs.com/eqnedit.php?latex=x_%7B0%7D%250) must travel through three RNN layers before the prediction is made. In terms of learning, this means that our gradients must be backpropagated through three RNN layers, while maintaining the flow of information through the network. If we compare this to a situation where we reverse our input sentence:



We now see that the distance between the true first word in our input sentence and the corresponding word in the output sentence is just 1 RNN layer. This means that gradients only need to be backpropagated one layer meaning the flow of information and the ability to learn is much greater for our network compared to when the distance between these two words was three layers.

If we were to calculate the total distances between the input words and their output counterparts for the reversed and non-reversed variants, we would see that they are the same. However, we have seen previously that our most important word in our output sentence is the first one. This is because words in our output sentences are dependent on the words that come before them. If we were to predict the first word in output sentence incorrectly then chances are the rest of the words in our sentences would be predicted incorrectly too. However, by predicting the first word correctly we maximize our chances of predicting the whole word correctly. Therefore, by minimizing the distance between the first word in our output sentence and its input counterpart, we can increase our model’s ability to learn this relationship, increase the chances of this prediction being correct and thus maximize the chances that our entire output sentence is predicted correctly.

1. With our tokenizers constructed, we next define the fields for our tokenization. Notice here how we append a start and end token to our sequences, so our model knows when to begin and end sequence input and output. We also convert all our input sentences into lower case for the sake of simplicity:

SOURCE = Field(tokenize = tokenize\_english,

init\_token = '<sos>',

eos\_token = '<eos>',

lower = True)

TARGET = Field(tokenize = tokenize\_german,

init\_token = '<sos>',

eos\_token = '<eos>',

lower = True)

1. With our fields defined, our tokenization becomes a simple one liner. The 30k dataset has built in training, validation and test sets that we can use for our model:

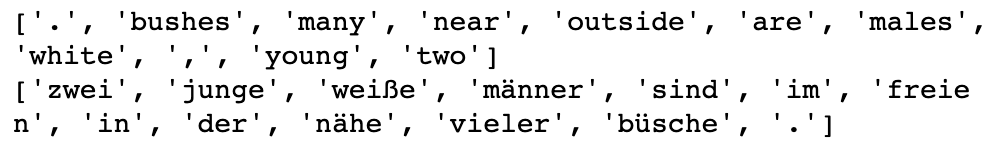
train\_data, valid\_data, test\_data = Multi30k.splits(exts = ('.en', '.de'), fields = (SOURCE, TARGET))

1. We can examine individual sentences using the examples property of our dataset objects. We see that the source (src) property contains our reversed input sentence in English and our target (trg) contains our non-reverse output sentence in German:

print(train\_data.examples[0].src)

print(train\_data.examples[0].trg)

Which gives us the following output:



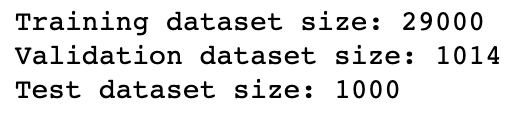
1. We can examine the size of each of our datasets. We see here that our training dataset consists of 29,000 examples and each of our validation and test sets consist of 1,014 and 1,000 examples respectively. In the past we have used 80%/20% splits of training to validation data. However, in instances like this where our input and output fields are very sparse and our training set is of limited size, it is often beneficial to train on as much data as is available:

print("Training dataset size: " + str(len(train\_data.examples)))

print("Validation dataset size: " + str(len(valid\_data.examples)))

print("Test dataset size: " + str(len(test\_data.examples)))

Which returns the following output:



1. Now, we can build our vocabularies and check their size. Our vocabularies should consist of every unique word found within our dataset. We see that our German vocabulary is considerably larger than our English vocabulary. Our vocabularies are significantly smaller than the true size of each vocabulary for each language (i.e.. every word in the English dictionary). Therefore, as our models will only be able to accurately translate words it has seen before, it is unlikely that our model will be able to generalize well to all possible sentences in the English language. This is why training models like this accurately requires extremely large NLP datasets (such as those Google has access to):

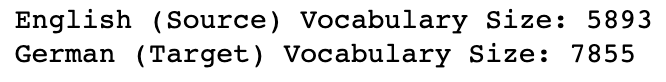
SOURCE.build\_vocab(train\_data, min\_freq = 2)

TARGET.build\_vocab(train\_data, min\_freq = 2)

print("English (Source) Vocabulary Size: " + str(len(SOURCE.vocab)))

print("German (Target) Vocabulary Size: " + str(len(TARGET.vocab)))

Which gives the following output:



1. Finally, we can create our data iterators from our datasets. As before we specify the usage of a CUDA enabled GPU if it is available on our system and specify our batch size.

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

batch\_size = 32

train\_iterator, valid\_iterator, test\_iterator = BucketIterator.splits(

(train\_data, valid\_data, test\_data),

batch\_size = batch\_size,

device = device)

Now, our data is preprocessed, we can begin to build the model itself.

**Building the Encoder**

We are ready to begin building our encoder:

1. We first begin by initializing our model by inheriting from our nn.Module class as with all our previous models. We initialize with a couple of parameters which we will define later; the number of dimensions in our hidden layers within our LSTM layers and the number of LSTM layers:

class Encoder(nn.Module):

def \_\_init\_\_(self, input\_dims, emb\_dims, hid\_dims, n\_layers, dropout):

super().\_\_init\_\_()

self.hid\_dims = hid\_dims

self.n\_layers = n\_layers

1. Next, we define our embedding layer within our encoder which is the length of the number of input dimensions and the depth of the number of embedding dimensions:

self.embedding = nn.Embedding(input\_dims, emb\_dims)

1. Next, we define our actual LSTM layer. This takes our embedded sentences from the embedding layer, maintains a hidden state of define length and consists of a number of layers (which we will define later as 2). We also implement dropout to apply regularization to our network:

self.rnn = nn.LSTM(emb\_dims, hid\_dims, n\_layers, dropout = dropout)

self.dropout = nn.Dropout(dropout)

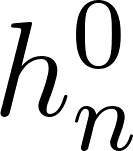
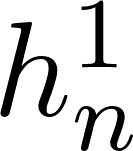
1. Then, we define the forward pass within our encoder. We apply the embeddings to our input sentences and apply dropout. Then we pass these embeddings through our LSTM layer which outputs our final hidden state which will then be used by our decoder to form our translated sentence.

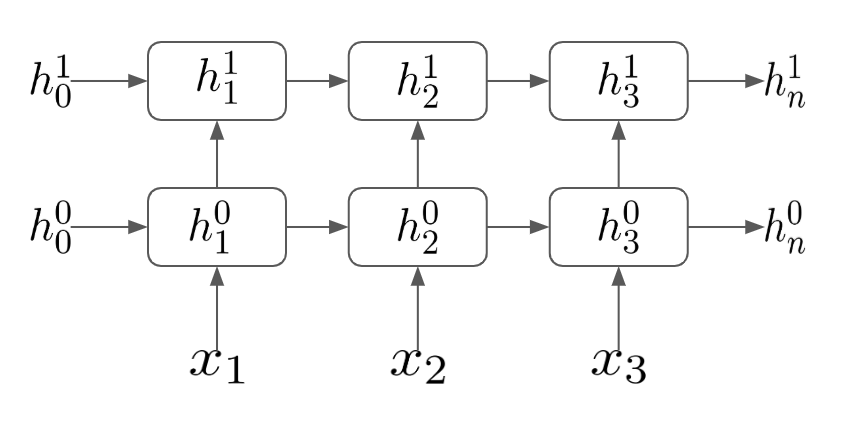
def forward(self, src):

embedded = self.dropout(self.embedding(src))

outputs, (h, cell) = self.rnn(embedded)

return h, cell

Our encoders will consist of two LSTM layers which means that our output will output two hidden states. This also means that our full LSTM layer with our encoder will look something like this, with our model outputting two hidden states [](https://www.codecogs.com/eqnedit.php?latex=h%5E%7B0%7D_%7Bn%7D%250) and [](https://www.codecogs.com/eqnedit.php?latex=h%5E%7B1%7D_%7Bn%7D%250) :



**Building the Decoder**

Our decoder will take the final hidden states from our encoder’s LSTM layer and translate this into an output sentence in another language. We start by initializing our decoder in almost exactly the same way as we did for the encoder. The only difference here is that we also add a fully connected linear layer. This layer will use our final hidden states from our LSTM in order to make predictions of the correct word in the sentence.

class Decoder(nn.Module):

def \_\_init\_\_(self, output\_dims, emb\_dims, hid\_dims, n\_layers, dropout):

super().\_\_init\_\_()

self.output\_dims = output\_dims

self.hid\_dims = hid\_dims

self.n\_layers = n\_layers

self.embedding = nn.Embedding(output\_dims, emb\_dims)

self.rnn = nn.LSTM(emb\_dims, hid\_dims, n\_layers, dropout = dropout)

self.fc\_out = nn.Linear(hid\_dims, output\_dims)

self.dropout = nn.Dropout(dropout)

Our forward pass is incredibly similar to that of our encoder, except with the addition of two key steps. We first unsqueeze our input from the previous layer in order to be the correct size for entry into our embedding layer. We also add a fully connected layer which takes the output hidden layer of our RNN layers and uses it to make a prediction of the next word in the sequence.

def forward(self, input, h, cell):

input = input.unsqueeze(0)

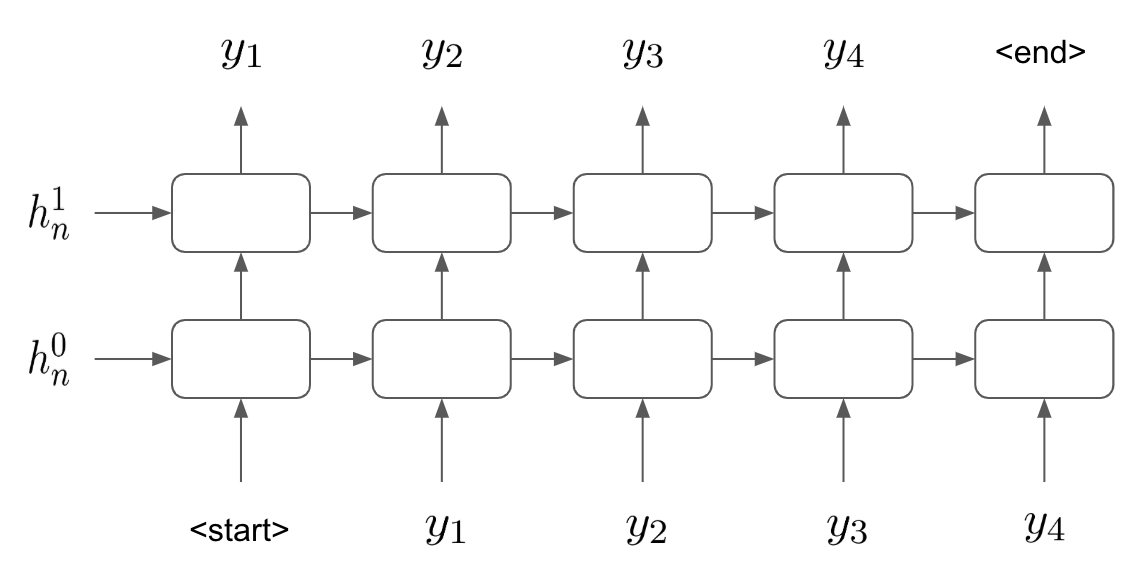
embedded = self.dropout(self.embedding(input))

output, (h, cell) = self.rnn(embedded, (h, cell))

pred = self.fc\_out(output.squeeze(0))

return pred, h, cell

Again, similar to our encoder, we use a 2-layer LSTM layer within our decoder. We take our final hidden state from our encoders and use these to generate the first word in our sequence $$y\_{1}$$. We then update our hidden state and use this and $$y\_{1}$$ to generate our next word $$y\_{2}$$, repeating this process until our model generates an end token. Our decoder looks something like this:



While we see that defining the encoder and decoders individually are not particularly complicated. However, when we combine these steps into one larger sequence to sequence model, things begin to get interesting:

**Constructing the Full Sequence to Sequence Model**

We must now stitch the two halves of our model together to produce the full sequence to sequence model:

1. We start by creating a new sequence to sequence class. This will allow us to pass our encoder and decoder to it as arguments:

class Seq2Seq(nn.Module):

def \_\_init\_\_(self, encoder, decoder, device):

super().\_\_init\_\_()

self.encoder = encoder

self.decoder = decoder

self.device = device

1. Next we create the forward method within our Seq2Seq class. This is arguably the most complicated part of the model. We combine our encoder with our decoder and use teacher forcing to help our model learn. We start by creating a tensor in which we still store our predictions. With initialize this as a Tensor of zeroes, but we still update this with our predictions as we make them. The shape of our tensor of zeroes will be the length of our target sentence, the width of our batch size and the depth of our target (German) vocabulary size:

def forward(self, src, trg, teacher\_forcing\_rate = 0.5):

batch\_size = trg.shape[1]

target\_length = trg.shape[0]

target\_vocab\_size = self.decoder.output\_dims

outputs = torch.zeros(target\_length, batch\_size, target\_vocab\_size).to(self.device)

1. We next feed our input sentence to our encoder to get the output hidden states:

h, cell = self.encoder(src)

1. Then, we must loop through our decoder model to generate an output prediction for each step in our output sequence. Our first element of our output sequence will always be the <start> token. Our target sequences already contain this as the first element, so we just set our initial input equal to this by taking the first element of the list:

input = trg[0,:]

1. Next, we loop through and make our predictions. We pass our hidden states (from the output of our encoder) to our decoder, along with our initial input (which is just the start token). This returns a prediction for all words in our sequence, however we are only interested in the word within our current step ie. the next word in the sequence. Note how we start our loop from 1 instead of 0 so our first prediction is the second word in the sequence (as the first word predicted will always be the start token).

This output consists of a vector of target vocabulary length, with prediction for each word within the vocabulary. We take the argmax to identify the actual word that is predicted by the model.

We then need to select our new input for the next step. We set our teacher forcing ratio to 50% which means 50% of the time we will use the prediction we just made as our next input into our decoder, and the other 50% of the time we will take the true target. As discussed before, this helps our model to learn much more rapidly than relying on just the model’s predictions.

We then continue this loop until we have a full prediction of each word in the sequence:

for t in range(1, target\_length):

output, h, cell = self.decoder(input, h, cell)

outputs[t] = output

top = output.argmax(1)

input = trg[t] if (random.random() < teacher\_forcing\_rate) else top

return outputs

1. Finally, we create an instance of our Seq2Seq model, ready to be trained. We initialize an encoder and a decoder with a selection of hyperparameters which can be changed to slightly alter the model:

input\_dimensions = len(SOURCE.vocab)

output\_dimensions = len(TARGET.vocab)

encoder\_embedding\_dimensions = 256

decoder\_embedding\_dimensions = 256

hidden\_layer\_dimensions = 512

number\_of\_layers = 2

encoder\_dropout = 0.5

decoder\_dropout = 0.5

1. We then create pass our encoder and decoder to our Seq2Seq model in order to create the complete model:

encod = Encoder(input\_dimensions, encoder\_embedding\_dimensions,\

hidden\_layer\_dimensions, number\_of\_layers, encoder\_dropout)

decod = Decoder(output\_dimensions, decoder\_embedding\_dimensions,\

hidden\_layer\_dimensions, number\_of\_layers, decoder\_dropout)

model = Seq2Seq(encod, decod, device).to(device)

Try experimenting with different parameters here and see how it affects the performance of the model. For instance, having a larger number of dimensions in your hidden layers may cause the model to train slower, although the overall final performance of the model may be better. Alternatively, the model may overfit. Very often it is a matter of experimentation to find the best performing model.

After fully defining our Seq2Seq model, we are now ready to begin training it.

**Training the Model**

Our model will begin initialized with weights of 0 across all parts of the model. While the model should theoretically be able to learn with no (zero) weights, it has been shown that initializing with random weights can help the model learn faster.

1. Here, we initialize our model with weights randomly samples from a normal distribution, with values between -0.1 and 0.1:

def initialize\_weights(m):

for name, param in m.named\_parameters():

nn.init.uniform\_(param.data, -0.1, 0.1)

model.apply(initialize\_weights)

1. Next, as with all our other models, we define our optimizer and loss function. We use cross entropy loss as we are performing multi-class classification (as opposed to binary cross-entropy loss for a binary classification).

optimizer = optim.Adam(model.parameters())

criterion = nn.CrossEntropyLoss(ignore\_index = TARGET.vocab.stoi[TARGET.pad\_token])

1. We next define the training process within a function called train(). We first set out model to train mode and set the epoch loss to 0:

def train(model, iterator, optimizer, criterion, clip):

model.train()

epoch\_loss = 0

1. We then loop through each batch within our training iterator and extract the sentence to be translated (src) and the correct translation of this sentence (trg). We then zero our gradients (to prevent gradient accumulation) and calculate the output of our model by passing our model function our inputs and outputs:

for i, batch in enumerate(iterator):

src = batch.src

trg = batch.trg

optimizer.zero\_grad()

output = model(src, trg)

1. Next, we need to calculate the loss of our model’s prediction by comparing our predicted output to the true, correct translated sentence. We reshape our output data and our target data using the shape and view functions in order to create two tensors which can be compared to calculate loss. We calculate the loss between our output and trg tensors and then back-propagate this loss back through the network:

output\_dims = output.shape[-1]

output = output[1:].view(-1, output\_dims)

trg = trg[1:].view(-1)

loss = criterion(output, trg)

loss.backward()

1. We then implement gradient clipping to prevent exploding gradients within our model, step our optimizer in order to perform the necessary parameter updates via gradient descent and finally add the loss of the batch to the epoch loss. This whole process is repeated for all the batches within a single training epoch, whereby the final averaged loss per batch is returned:

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), clip)

optimizer.step()

epoch\_loss += loss.item()

return epoch\_loss / len(iterator)

1. Similarly, we create a similar function called evaluate(). This function will calculate the loss of our validation data across the network in order to evaluate how our model performs when translating data it hasn’t seen before. The function is almost identical to our train() function, with the exception of the fact that we switch to evaluation mode:

model.eval()

1. And as we don’t perform any updates of our weights, we make sure to implement no\_grad mode:

with torch.no\_grad():

1. The only other difference is that we make sure to turn off teacher forcing when in evaluation mode. We wish to assess our model’s performance on unseen data and enabling teacher forcing would use our correct (target) data to help our model make better predictions. We want our model to be able to make perfect, unaided predictions, so we make sure to evaluate with teacher forcing switched off:

output = model(src, trg, 0)

1. Finally, we need to create a training loop, within which our train() and evaluate() functions are called. We begin by defining how many epochs we wish to train for and our maximum gradient (for use with gradient clipping). We also set our lowest validation loss to infinity. This will be used later to select our best performing model:

epochs = 10

grad\_clip = 1

lowest\_validation\_loss = float('inf')

1. We then loop through each of our epochs and within each epoch, calculate our training and validation loss using our train() and evaluate() functions. We also time how long this takes by calling time.time() before and after the training:

for epoch in range(epochs):

start\_time = time.time()

train\_loss = train(model, train\_iterator, optimizer, criterion, grad\_clip)

valid\_loss = evaluate(model, valid\_iterator, criterion)

end\_time = time.time()

1. Next, for each epoch we determine whether the model we just trained is the best performing model we have seen thus far. If our model performs the best on our validation data (ie. if the validation loss is the lowest, we have seen so far), we save our model:

if valid\_loss < lowest\_validation\_loss:

lowest\_validation\_loss = valid\_loss

torch.save(model.state\_dict(), 'seq2seq.pt')

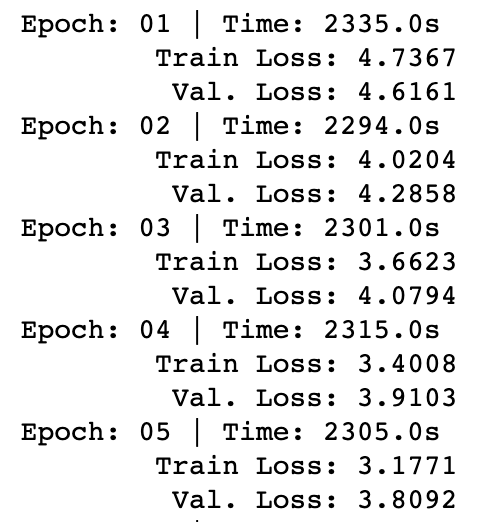
1. Finally, we simply print our output:

print(f'Epoch: {epoch+1:02} | Time: {np.round(end\_time-start\_time,0)}s')

print(f'\tTrain Loss: {train\_loss:.4f}')

print(f'\t Val. Loss: {valid\_loss:.4f}')

If our training is working correctly, we should see training loss decrease over time like so:



Here, we see that both our training and validation loss appear to be falling over time. We can continue to train our model for a number of epochs, ideally until the validation loss reaches its lowest possible value. Now, we can then evaluate our best-performing model to see how well it performs when making actual translations.

**Evaluating the Model**

In order to evaluate our model, we will take our test set of data and run our English sentences through our model to obtain a prediction of the translation in German. We will then be able to compare this to the true prediction in order to see if our model is making accurate predictions.

1. We start by creating a function translate() which is functionally identical to our evaluate() function we created to calculate the loss over our validation set. However, this time we are not concerned with the loss of our model, but rather the predicted output. We pass the model our source and target sentences, and make sure we turn teacher forcing off, so our model does not use these to make predictions. We then take our model’s predictions and use an argmax function to determine the index of the word that our model predicted for each word in our predicted output sentence:

output = model(src, trg, 0)

preds = torch.tensor([[torch.argmax(x).item()] for x in output])

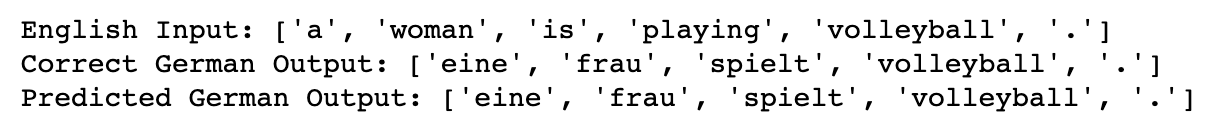
1. Then, we can use this index to obtain the actual predicted word from our German vocabulary. We finally compare the English input to our model with the correct German sentence and the predicted German sentence. Note that here we use [1:-1] to drop the start and end tokens from our predictions and we reverse the order of our English input (as the input sentences were reversed before they were fed into the model.

print('English Input: ' + str([SOURCE.vocab.itos[x] for x in src][1:-1][::-1]))

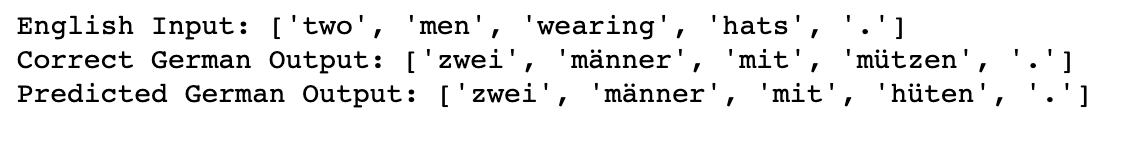
print('Correct German Output: ' + str([TARGET.vocab.itos[x] for x in trg][1:-1]))

print('Predicted German Output: ' + str([TARGET.vocab.itos[x] for x in preds][1:-1]))

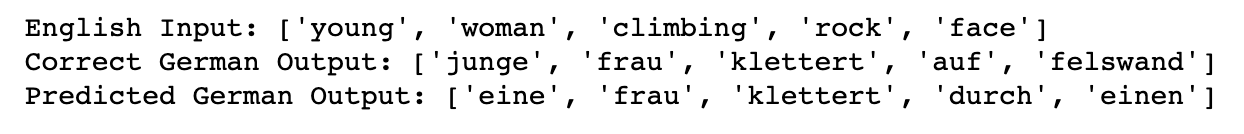
By doing this, we are able to compare our predicted output with the correct output to assess if our model is able to make accurate predictions. We can see from our model’s predictions that our model is able to translate English sentences into German, albeit far from perfectly. Some of our model’s predictions are exactly the same as the target data, showing that our model translated these sentences perfectly:



In other instances, our model is off by a single word. In this case out model predicts the word hüten instead of mützen, however hüten is actually an acceptable translation of mützen, however the words may not be semantically identical



We also see examples that seem mistranslated. In the example below, the English equivalent of the German sentence that we predicted is “A woman climbs through one” which is not equivalent to “Young woman climbing rock face”. However, the fact that the model has still managed to translate key elements of the English sentence (ie. woman and climbing):



We see that although our model clearly makes a decent attempt at translating English into German, however it is clearly far from perfect and makes several mistakes. It certainly would not be able to fool a native German speaker! We will discuss a couple of ways by which we could improve our sequence to sequence translation model.

**Next Steps**

While we have shown our sequence to sequence model to be effective at performing language translation, the model we trained from scratch is not a perfect translator by any means. This is in part due to the relatively small size of our training data. We trained our model on a set of 29,000 English/German sentences. While this might seem very large, in order to train a perfect model, we would require a training set of several orders of magnitude larger.

In theory, we would require several examples of each word in the entire English and German languages for our model to truly understand its context and meaning. For context, our 29,000 English sentences in our training set consisted of just 6,000 unique words. The average vocabulary of an English speaker is said to be between 20,000 and 30,000 words, which gives us an idea of just how many examples sentences we would need to train a model that performs perfectly. This is probably why the most accurate translation tools are owned by companies with access to vast amounts of language data (i.e.. Google).

**Summary**

In this chapter, we covered how to build sequence-to-sequence models from scratch. We learned how to code up our encoder and decoder components individually and how to integrate them into a single model that is able to translate sentences from one language to another.

Although our sequence to sequence model, consisting of an encoder and decoder, is useful for sequence translation, it is no longer state of the art. In the last few years, combining sequence to sequence models with attention models in order to achieve state of the art performance.

In our next chapter, we will discuss how attention networks can be used in the context of sequence to sequence learning and show how we can use both of these techniques to build a chat bot.