

CS 563: Natural Language Processing

IIT PATNA

Assignment-4: Neural Machine Translation

Preparing Data

First up, importing all the necessary libraries. The main ones we'll be using are:

- [PyTorch](#) for creating the models
- [spaCy](#) to assist in the tokenization of the data
- [torchtext](#) to provide some helper functions
- [datasets](#) to load and manipulate our data
- [evaluate](#) for calculating metrics

```
import torch
import torch.nn as nn
import torch.optim as optim
import random
import numpy as np
import spacy
import datasets
import torchtext
import tqdm
import evaluate
```

```
/home/ben/miniconda3/envs/main/lib/python3.9/site-
packages/thinc/compat.py:36: UserWarning: 'has_mps' is deprecated, please use
'torch.backends.mps.is_built()'
  hasattr(torch, "has_mps")
/home/ben/miniconda3/envs/main/lib/python3.9/site-
packages/thinc/compat.py:37: UserWarning: 'has_mps' is deprecated, please use
'torch.backends.mps.is_built()'
  and torch.has_mps # type: ignore[attr-defined]
addCodeaddText
```

We'll set all possible random seeds for deterministic results.

```
seed = 1234

random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.backends.cudnn.deterministic = True
```

Dataset

Next, we'll load our dataset using the `datasets` library. When using the `load_dataset` function we pass the name of the dataset, `bentrevett/multi30k`.

The dataset we'll be using is a subset of the [Multi30k dataset](#), which is hosted [here](#) on the HuggingFace dataset hub. This subset has ~30,000 parallel English and German sentences obtained using the task 1 raw data from [here](#). We use the "2016" versions of the test sets.

```
dataset = datasets.load_dataset("bentrevett/multi30k")
```

We can see the `dataset` object (a `DatasetDict`) contains the train, validation and test splits, the amount of examples in each split, and the features in each split ("en" and "de").

```
DatasetDict({
  train: Dataset({
    features: ['en', 'de'],
    num_rows: 29000
  })
  validation: Dataset({
    features: ['en', 'de'],
    num_rows: 1014
  })
  test: Dataset({
    features: ['en', 'de'],
    num_rows: 1000
  })
})
```

For convenience, we create a variable for each split. Each being a `Dataset` object.

`addCodeaddTe`

```
train_data, valid_data, test_data = (
    dataset["train"],
    dataset["validation"],
    dataset["test"],
)
```

We can index into each `Dataset` to view an individual example. Each example has two features: "en" and "de", which are the parallel English and German sentences.

```
train_data[0]
```

```
{'en': 'Two young, White males are outside near many bushes.',  
'de': 'Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.'}  
addCodeaddText
```

Tokenizers

Tokenizers

Next, we'll load the spaCy models that contain the tokenizers.

A tokenizer is used to turn a string into a list of tokens that make up that string, e.g. "good morning!" becomes ["good", "morning", "!"]. We'll start talking about the sentences being a sequence of tokens from now, instead of saying they're a sequence of words. What's the difference? Well, "good" and "morning" are both words and tokens, but "!" is not a word. We could say "!" is punctuation, but the term token is more general and covers: words, punctuation, numbers and any special symbols.

spaCy has model for each language ("de_core_news_sm" for German and "en_core_web_sm" for English) which need to be loaded so we can access the tokenizer of each model.

Note: the models must first be downloaded using the following on the command line:

```
python -m spacy download en_core_web_sm  
python -m spacy download de_core_news_sm
```

We load the models as such:

```
en_nlp = spacy.load("en_core_web_sm")  
de_nlp = spacy.load("de_core_news_sm")
```

```
/home/ben/miniconda3/envs/main/lib/python3.9/site-packages/spacy/util.py:910: UserWarning: [W095] Model  
'en_core_web_sm' (3.5.0) was trained with spaCy v3.5.0 and may not be 100% compatible with the current version (3.7.2).  
If you see errors or degraded performance, download a newer compatible model or retrain your custom model with the  
current spaCy version. For more details and available updates, run: python -m spacy validate warnings.warn(warn_msg)  
/home/ben/miniconda3/envs/main/lib/python3.9/site-packages/spacy/util.py:910: UserWarning: [W095] Model  
'de_core_news_sm' (3.5.0) was trained with spaCy v3.5.0 and may not be 100% compatible with the current version (3.7.2).  
If you see errors or degraded performance, download a newer compatible model or retrain your custom model with the  
current spaCy version. For more details and available updates, run: python -m spacy validate warnings.warn(warn_msg)
```

We can call the tokenizer for each spaCy model using the `.tokenizer` method, which accepts a string and returns a sequence of Token objects. We can get the string from the token object using the `text` attribute.

```
nizer(string)]string = "What a lovely day it is today!"
```

```
[token.text for token in en_nlp.tokenize(string)]  
['What', 'a', 'lovely', 'day', 'it', 'is', 'today', '!']
```

Next, we'll write a function used to apply the tokenizer to all of the examples in each data split, as well as apply some other processing.

This function takes in an example from the `Dataset` object, applies the tokenizers English and German spaCy models, trims the list of tokens to a maximum length, optionally converts each token to lowercase, and then appends the start of sequence and end of sequence tokens to the beginning and end of the list of tokens.

This function will be used with the `map` method from a `Dataset`, which needs to return a dictionary containing the names of the features in each example where the outputs are stored. As the output feature names "en_tokens" and "de_tokens" are not already contained in the example (where we only have "en" and "de" features), this will create two new features in each example.

```
def tokenize_example(example, en_nlp, de_nlp, max_length, lower, sos_token, eos_token):
    en_tokens = [token.text for token in en_nlp.tokenizer(example["en"])][:max_length]
    de_tokens = [token.text for token in de_nlp.tokenizer(example["de"])][:max_length]
    if lower:
        en_tokens = [token.lower() for token in en_tokens]
        de_tokens = [token.lower() for token in de_tokens]
    en_tokens = [sos_token] + en_tokens + [eos_token]
    de_tokens = [sos_token] + de_tokens + [eos_token]
    return {"en_tokens": en_tokens, "de_tokens": de_tokens}
```

We apply the `tokenize_example` function using the `map` method as below.

The `example` argument is implied, however all additional arguments to the `tokenize_example` function need to be stored in a dictionary and passed to the `fn_kwargs` argument of `map`.

Here, we're trimming all sequences to a maximum length of 1000 tokens, converting each token to lower case, and using `<sos>` and `<eos>` as the start and end of sequence tokens, respectively.

```
max_length = 1_000
lower = True
sos_token = "<sos>"
eos_token = "<eos>"

fn_kwargs = {
    "en_nlp": en_nlp,
    "de_nlp": de_nlp,
    "max_length": max_length,
```

```

    "lower": lower,
    "sos_token": sos_token,
    "eos_token": eos_token,
}

train_data = train_data.map(tokenize_example, fn_kwargs=fn_kwargs)
valid_data = valid_data.map(tokenize_example, fn_kwargs=fn_kwargs)
test_data = test_data.map(tokenize_example, fn_kwargs=fn_kwargs)
Map: 0%|          | 0/1014 [00:00<?, ? examples/s]
Map: 0%|          | 0/1000 [00:00<?, ? examples/s]

```

We can now look at an example, confirming the two new features have been added; both of which are lowercased list of strings with the start/end of sequence tokens appended.

```

train_data[0]
{'en': 'Two young, White males are outside near many bushes.',
 'de': 'Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.',
 'en_tokens': ['<sos>',
 'two',
 'young',
 ',',
 'white',
 'males',
 'are',
 'outside',
 'near',
 'many',
 'bushes',
 ':',
 '<eos>'],
 'de_tokens': ['<sos>',
 'zwei',
 'junge',
 'weiße',
 'männer',
 'sind',
 'weiße',
 'männer',

 '<eos>sind',
 'im',
 'freien',

```

```
'in',  
'der',  
'nähe',  
'vieler',  
'büsche',  
'',  
>']}]}
```

Vocabularies

Next, we'll build the vocabulary for the source and target languages. The vocabulary is used to associate each unique token in our dataset with an index (an integer), e.g. "hello" = 1, "world" = 2, "bye" = 3, "hates" = 4, etc. When feeding text data to our model, we convert the string into tokens and then the tokens into numbers using the vocabulary as a look up table, e.g. "hello world" becomes ["hello", "world"] which becomes [1, 2] using the example indices given. We do this as neural networks cannot operate on strings, only numerical values.

We create the vocabulary (one for each language) from our datasets using the `build_vocab_from_iterator` function, provided by `torchtext`, which accepts an iterator where each item is a list of tokens. It then counts up the number of unique tokens and assigns each a numerical value.

"unknown token", denoted by `<unk>`, which is given its own index (usually index zero). All unknown tokens are replaced by `<unk>`, even if the tokens are different, i.e. if the tokens "gilgamesh" and "enkidu" were both not within our vocabulary, then the string "gilgamesh hates enkidu" gets tokenized to ["gilgamesh", "hates", "enkidu"] and then becomes [0, 24, 0] (where "hates" has the index 24).

. Hence, when creating our vocabularies with `build_vocab_from_iterator`, we use the `min_freq` argument to not create an index for tokens which appear less than `min_freq` times in our training set. In other words, when using the vocabulary, any token which does not appear at least twice in our training set will get replaced by the unknown token index when converting tokens to indices.

We also use the `specials` argument of `build_vocab_from_iterator` to pass special tokens. These are tokens which we want to add to the vocabulary but do not necessarily appear in our tokenized examples. These special tokens will appear first in the vocabulary. We've already discussed the `unk_token`, `sos_token`, and `eos_token`. The final special token is the `pad_token`, denoted by `<pad>`.

The majority of our sentences are not the same length, but we can solve this by "padding" (adding `<pad>` tokens) the tokenized version of each sentence in a batch until they all have

equal tokens to the longest sentence in the batch. For example, if we had two sentences: "I love pizza" and "I hate music videos". They would be tokenized to something like: ["i", "love", "pizza"] and ["i", "hate", "music", "videos"]. The first sequence of tokens would then be padded to ["i", "love", "pizza", "<pad>"]. Both sequences could then be converted to indexes using the vocabulary.

That was a lot of information, but luckily torchtext handles all the fuss of building the vocabulary.

```
min_freq = 2
unk_token = "<unk>"
pad_token = "<pad>"

special_tokens = [
    unk_token,
    pad_token,
    sos_token,
    eos_token,
]

en_vocab = torchtext.vocab.build_vocab_from_iterator(
    train_data["en_tokens"],
    min_freq=min_freq,
    specials=special_tokens,
)

de_vocab = torchtext.vocab.build_vocab_from_iterator(
    train_data["de_tokens"],
    min_freq=min_freq,
    specials=special_tokens,
)
```

We can get the first ten tokens in our vocabulary (indices 0 to 9) using the `get_itos` method, where `itos` = "int to string", which returns a list of tokens.

```
en_vocab.get_itos()[:10]
['<unk>', '<pad>', '<sos>', '<eos>', 'a', '.', 'in', 'the', 'on', 'man']
en_vocab.get_itos()[9]
'man'
```

This is because the vocabularies were effectively created from different data (one English and one German) even though they were from the same examples. The indices given to tokens that are not special tokens are ordered from most frequent to least frequent (though still appearing at least `min_freq` times).

```
de_vocab.get_itos()[:10]
```



```
['<unk>', '<pad>', '<sos>', '<eos>', '.', 'ein', 'einem', 'in', 'eine', ',']
```

We can get the index from a given token using the `get_stoi` (`stoi` = "string to int") method.

```
en_vocab.get_stoi()["the"]
```

```
7
```

we can just use the vocabulary as a dictionary and pass the token to get the index. Note that this doesn't work the other way around, i.e. `en_vocab[7]` does not work.

```
en_vocab["the"]
```

```
7
```

We can see that our training data had around 2000 more German tokens (that appeared at least twice) than the English data.

```
len(en_vocab), len(de_vocab)
```

```
(5893, 7853)
```

We can also use the `in` keyword to get a boolean indicating if a token is in the vocabulary.

```
"the" in en_vocab
```

```
True
```

This means that no tokens containing any uppercase characters appear in our vocabulary.

```
"The" in en_vocab
```

```
False
```

One quirk of the `torchtext` vocabulary class is that you have to manually set what value you want your vocabulary to return when you try and get the index of an out-of-vocabulary token. If you have not set this value, then you will receive an error! This is so you can set your vocabulary to return any value when trying to get the index of a token not in the vocabulary, even something like -100.

```
#en_vocab["The"]
```

We also get the index of our `<pad>` token, as we'll use it later

```
assert en_vocab[unk_token] == de_vocab[unk_token]
```

```
assert en_vocab[pad_token] == de_vocab[pad_token]
```

```
unk_index = en_vocab[unk_token]
```

```
pad_index = en_vocab[pad_token]
```

Using the `set_default_index` method we can set what value is returned when we try and get the index of a token outside of our vocabulary. In this case, the index of the unknown token, `<unk>`.

```
en_vocab.set_default_index(unk_index)
```

```
de_vocab.set_default_index(unk_index)
```

we can happily get indexes of out of vocabulary tokens until our heart is content!

```
en_vocab["The"]  
0
```

And we can get the token corresponding to that index to prove it's the <unk> token.

```
en_vocab.get_itos()[0]  
'<unk>'
```

useful feature of the vocabulary is the `lookup_indices` method. This takes in a list of tokens and returns a list of indices. In the below example we can see the token "crime" does not exist in our vocabulary so is converted to the index of the <unk> token, zero, which we passed to the `set_default_index` method.

```
tokens = ["i", "love", "watching", "crime", "shows"]  
en_vocab.lookup_indices(tokens)  
[956, 2169, 173, 0, 821]
```

we can use the `lookup_tokens` method to convert a list of indices back into tokens using the vocabulary. Notice how the original "crime" token is now an <unk> token. There is no way to tell what the original sequence of tokens was.

```
en_vocab.lookup_tokens(en_vocab.lookup_indices(tokens))  
['i', 'love', 'watching', '<unk>', 'shows']
```

we've now got the gist of how the `torchtext.Vocab` class works. Time to put it into action!

Just like our `tokenize_example`, we create a `numericalize_example` function which we'll use with the `map` method of our dataset. This will "numericalize" (a fancy way of saying convert tokens to indices) our tokens in each example using the vocabularies and return the result into new "en_ids" and "de_ids" features.

```
def numericalize_example(example, en_vocab, de_vocab):  
    en_ids = en_vocab.lookup_indices(example["en_tokens"])  
    de_ids = de_vocab.lookup_indices(example["de_tokens"])  
    return {"en_ids": en_ids, "de_ids": de_ids}
```

apply the `numericalize_example` function, passing our vocabularies in the `fn_kwargs` dictionary to the `fn_kwargs` argument.

```
fn_kwargs = {"en_vocab": en_vocab, "de_vocab": de_vocab}
```

```
train_data = train_data.map(numericalize_example, fn_kwargs=fn_kwargs)
valid_data = valid_data.map(numericalize_example, fn_kwargs=fn_kwargs)
test_data = test_data.map(numericalize_example, fn_kwargs=fn_kwargs)
```

```
Map: 0%|          | 0/1014 [00:00<?, ? examples/s]
Map: 0%|          | 0/1000 [00:00<?, ? examples/s]
```

we can see that it has the two new features: "en_ids" and "de_ids", both a list of integers representing their indices in the respective vocabulary.

```
train_data[0]
{'en': 'Two young, White males are outside near many bushes.',
 'de': 'Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.',
 'en_tokens': ['<sos>',
 'two',
 'young',
 ',',
 'white',
 'males',
 'are',
 'outside',
 'near',
 'many',
 'bushes',
 ',',
 '<eos>'],
 'de_tokens': ['<sos>',
 'zwei',
 'junge',
 'weiße',
 'männer',
 'sind',
 'im',
 'freien',
 'in',
 'der',
 'nähe',
 'vieler',
 'büsche',
 ',',
 '<eos>'],
 'en_ids': [2, 16, 24, 15, 25, 778, 17, 57, 80, 202, 1312, 5, 3],
 'de_ids': [2, 18, 26, 253, 30, 84, 20, 88, 7, 15, 110, 7647, 3171, 4, 3]}
```

```
'en': 'Two young, White males are outside near many bushes.',
'de': 'Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.',
'en_tokens': ['<sos>',
'two',
'young',
'',
'white',
'males',
'are',
'outside',
'near',
'many',
'bushes',
'',
'<eos>'],
'de_tokens': ['<sos>',
'zwei',
'junge',
'weiße',
'männer',
'sind',
'im',
'freien',
'in',
'der',
'nähe',
'vieler',
'büsche',
'',
'<eos>']}]}
```

We can also check this using the type built-in function on one of the features.

```
type(train_data[0]["en_ids"])
torch.Tensor
```

Data Loaders:

The final step of preparing the data is to create the data loaders. These can be iterated upon to return a batch of data, each batch being a dictionary containing the numericalized English and German sentences (which have also been padded) as PyTorch tensors.

First, we need to create a function that collates, i.e. combines, a batch of examples into a batch. The `collate_fn` below takes a "batch" as input (a list of examples), we then separate

out the English and German indices for each example in the batch, and pass each one to the `pad_sequence` function. `pad_sequence` takes a list of tensors, pads each one to the length of the longest tensor using the `padding_value` (which we set to `pad_index`, the index of our <pad> token) and then returns a [max length, batch size] shaped tensor, where batch size is the number of examples in the batch and max length is the length of the longest tensor in the batch. We put each tensor into a dictionary and then return it.

The `get_collate_fn` takes in the padding token index and returns the `collate_fn` defined inside it. This technique, of defining a function inside another and returning it, is known as a [closure](#). It allows the `collate_fn` to continually use the value of `pad_index` it was created with without creating a class or using global variables.

```
def get_collate_fn(pad_index):
    def collate_fn(batch):
        batch_en_ids = [example["en_ids"] for example in batch]
        batch_de_ids = [example["de_ids"] for example in batch]
        batch_en_ids = nn.utils.rnn.pad_sequence(batch_en_ids, padding_value=pad_index)
        batch_de_ids = nn.utils.rnn.pad_sequence(batch_de_ids, padding_value=pad_index)
        batch = {
            "en_ids": batch_en_ids,
            "de_ids": batch_de_ids,
        }
        return batch

    return collate_fn
```

Next, we write the functions which give us our data loaders created using PyTorch's `DataLoader` class.

`get_data_loader` is created using a `Dataset`, the batch size, the padding token index (which is used for creating the batches in the `collate_fn`, and a boolean deciding if the examples should be shuffled at the time the data loader is iterated over.

The batch size defines the maximum amount of examples within a batch. If the length of the dataset is not evenly divisible by the batch size then the last batch will be smaller.

```
def get_data_loader(dataset, batch_size, pad_index, shuffle=False):
    collate_fn = get_collate_fn(pad_index)
    data_loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=batch_size,
        collate_fn=collate_fn,
        shuffle=shuffle,
    )
    return data_loader
```

we create our data loaders.

To reduce the training time, we generally want to use the largest batch size possible. When using a GPU, this means using the largest batch size that will fit in GPU memory.

Shuffling of data makes training more stable and potentially improves the final performance of the model, however only needs to be done on the training set. The metrics calculated for the validation and test set will be the same no matter what order the data is in.

```
batch_size = 128

train_data_loader = get_data_loader(train_data, batch_size, pad_index, shuffle=True)
valid_data_loader = get_data_loader(valid_data, batch_size, pad_index)
test_data_loader = get_data_loader(test_data, batch_size, pad_index)
```

Building the Model:

We'll be building our model in three parts. The encoder, the decoder and a seq2seq model that encapsulates the encoder and decoder and will provide a way to interface with each.

Encoder

First, the encoder, a 2 layer LSTM. The paper we are implementing uses a 4-layer LSTM, but in the interest of training time we cut this down to 2-layers. The concept of multi-layer RNNs is easy to expand from 2 to 4 layers.

For a multi-layer RNN, the input sentence, X , after being embedded goes into the first (bottom) layer of the RNN and hidden states, $H=\{h_1, h_2, \dots, h_T\}$, output by this layer are used as inputs to the RNN in the layer above. Thus, representing each layer with a superscript, the hidden states in the first layer are given by:

$$h_{1t} = \text{EncoderRNN}_1(e(x_t), h_{1t-1})$$

The hidden states in the second layer are given by:

$$h_{2t} = \text{EncoderRNN}_2(h_{1t}, h_{2t-1})$$

Using a multi-layer RNN also means we'll also need an initial hidden state as input per layer, h_{10} , and we will also output a context vector per layer, z_l .

Without going into too much detail about LSTMs (see [this](#) blog post to learn more about them), all we need to know is that they're a type of RNN which instead of just taking in a hidden state and returning a new hidden state per time-step, also take in and return a *cell state*, c_t , per time-step.

$$h_t(h_t, c_t) = \text{RNN}(e(x_t), h_{t-1}) = \text{LSTM}(e(x_t), h_{t-1}, c_{t-1})$$

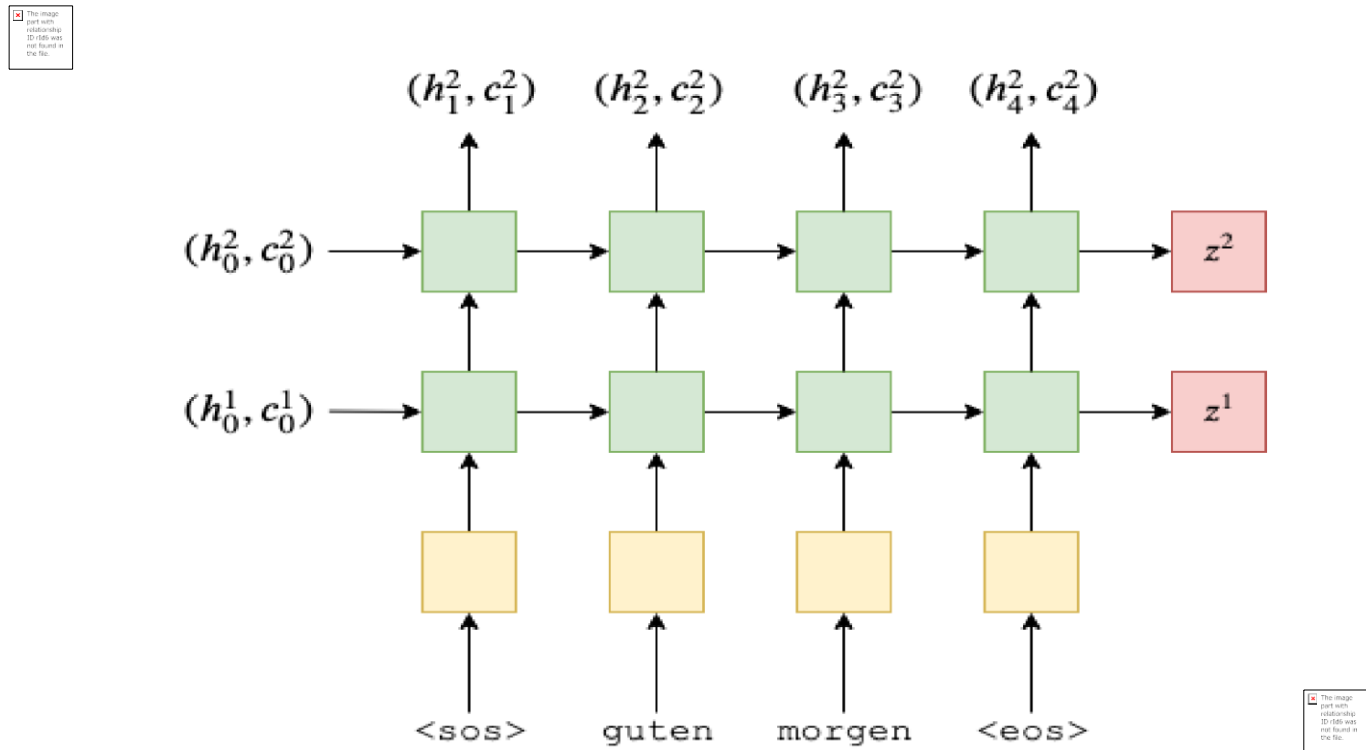
We can just think of c_t as another type of hidden state. Similar to h_{10} , c_{10} will be initialized to a tensor of all zeros. Also, our context vector will now be both the final hidden state and the final cell state, i.e. $z_l = (h_{1T}, c_{1T})$.

Extending our multi-layer equations to LSTMs, we get

$$(h_{1t}, c_{1t})(h_{2t}, c_{2t}) = \text{EncoderLSTM}_1(e(x_t), (h_{1t-1}, c_{1t-1})) = \text{EncoderLSTM}_2(h_{1t}, (h_{2t-1}, c_{2t-1}))$$

Note how only our hidden state from the first layer is passed as input to the second layer, and not the cell state.

So our encoder looks something like this:



Create this in code by making an Encoder module, which requires we inherit from `torch.nn.Module` and use the `super().__init__()` as some boilerplate code. The encoder takes the following arguments:

- `input_dim` is the size/dimensionality of the one-hot vectors that will be input to the encoder. This is equal to the input (source) vocabulary size.
- `embedding_dim` is the dimensionality of the embedding layer. This layer converts the one-hot vectors into dense vectors with `embedding_dim` dimensions.
- `hidden_dim` is the dimensionality of the hidden and cell states.
- `n_layers` is the number of layers in the RNN.

The RNN returns: outputs (the top-layer hidden state for each time-step), hidden (the final hidden state for each layer, `hT`, stacked on top of each other) and cell (the final cell state for each layer, `cT`, stacked on top of each other).

As we only need the final hidden and cell states (to make our context vector), forward only returns hidden and cell.

The sizes of each of the tensors is left as comments in the code. In this

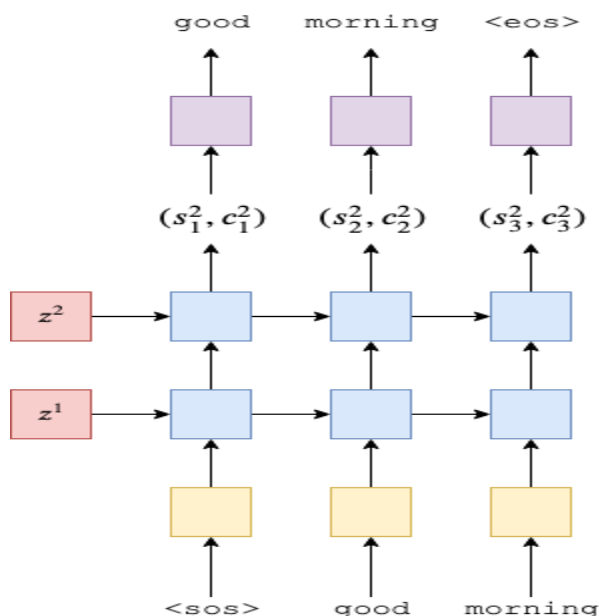
implementation `n_directions` will always be 1, however note that bidirectional RNNs will have `n_directions` as 2

```
class Encoder(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, n_layers, dropout):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        # src = [src length, batch size]
        embedded = self.dropout(self.embedding(src))
        # embedded = [src length, batch size, embedding dim]
        outputs, (hidden, cell) = self.rnn(embedded)
        # outputs = [src length, batch size, hidden dim * n directions]
        # hidden = [n layers * n directions, batch size, hidden dim]
        # cell = [n layers * n directions, batch size, hidden dim]
        # outputs are always from the top hidden layer
        return hidden, cell
```

Decoder

Next, we'll build our decoder, which will also be a 2-layer (4 in the paper) LSTM.



The Decoder class does a single step of decoding, i.e. it outputs single token per time-step. The first layer will receive a hidden and cell state from the previous time-step, (s_{1t-1}, c_{1t-1}) ,

and feeds it through the LSTM with the current embedded token, y_t , to produce a new hidden and cell state, (s_{1t}, c_{1t}) . The subsequent layers will use the hidden state from the layer below, s_{l-1t} , and the previous hidden and cell states from their layer, (s_{lt-1}, c_{lt-1}) . This provides equations very similar to those in the encoder.

$$(s_{1t}, c_{1t}) = \text{DecoderLSTM1}(d(y_t), (s_{1t-1}, c_{1t-1})) \quad (s_{2t}, c_{2t}) = \text{DecoderLSTM2}(s_{1t}, (s_{2t-1}, c_{2t-1}))$$

Remember that the initial hidden and cell states to our decoder are our context vectors, which are the final hidden and cell states of our encoder from the same layer, i.e. $(s_{l0}, c_{l0}) = z_l = (h_{lT}, c_{lT})$.

We then pass the hidden state from the top layer of the RNN, s_{Lt} , through a linear layer, f , to make a prediction of what the next token in the target (output) sequence should be, y^{t+1} .

$$\hat{y}^{t+1} = f(s_{Lt})$$

The arguments and initialization are similar to the Encoder class, except we now have an `output_dim` which is the size of the vocabulary for the output/target language. There is also the addition of the Linear layer, used to make the predictions from the top layer hidden state.

We then return the prediction, the new hidden state and the new cell state.

`nn.LSTMCell` is just a single cell and `nn.LSTM` is a wrapper around potentially multiple cells. Using the `nn.LSTMCell` in this case would mean we don't have to unsqueeze to add a fake sequence length dimension, but we would need one `nn.LSTMCell` per layer in the decoder and to ensure each `nn.LSTMCell` receives the correct initial hidden state from the encoder. All of this makes the code less concise -- hence the decision to stick with the regular `nn.LSTM`.

```
class Decoder(nn.Module):
    def __init__(self, output_dim, embedding_dim, hidden_dim, n_layers, dropout):
        super().__init__()
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(output_dim, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout)
        self.fc_out = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell):
        # input = [batch size]
        # hidden = [n layers * n directions, batch size, hidden dim]
        # cell = [n layers * n directions, batch size, hidden dim]
        # n directions in the decoder will both always be 1, therefore:
        # hidden = [n layers, batch size, hidden dim]
        # context = [n layers, batch size, hidden dim]
        input = input.unsqueeze(0)
```

```

# input = [1, batch size]
embedded = self.dropout(self.embedding(input))
# embedded = [1, batch size, embedding dim]
output, (hidden, cell) = self.rnn(embedded, (hidden, cell))
# output = [seq length, batch size, hidden dim * n directions]
# hidden = [n layers * n directions, batch size, hidden dim]
# cell = [n layers * n directions, batch size, hidden dim]
# seq length and n directions will always be 1 in this decoder, therefore:
# output = [1, batch size, hidden dim]
# hidden = [n layers, batch size, hidden dim]
# cell = [n layers, batch size, hidden dim]
prediction = self.fc_out(output.squeeze(0))
# prediction = [batch size, output dim]
return prediction, hidden, cell

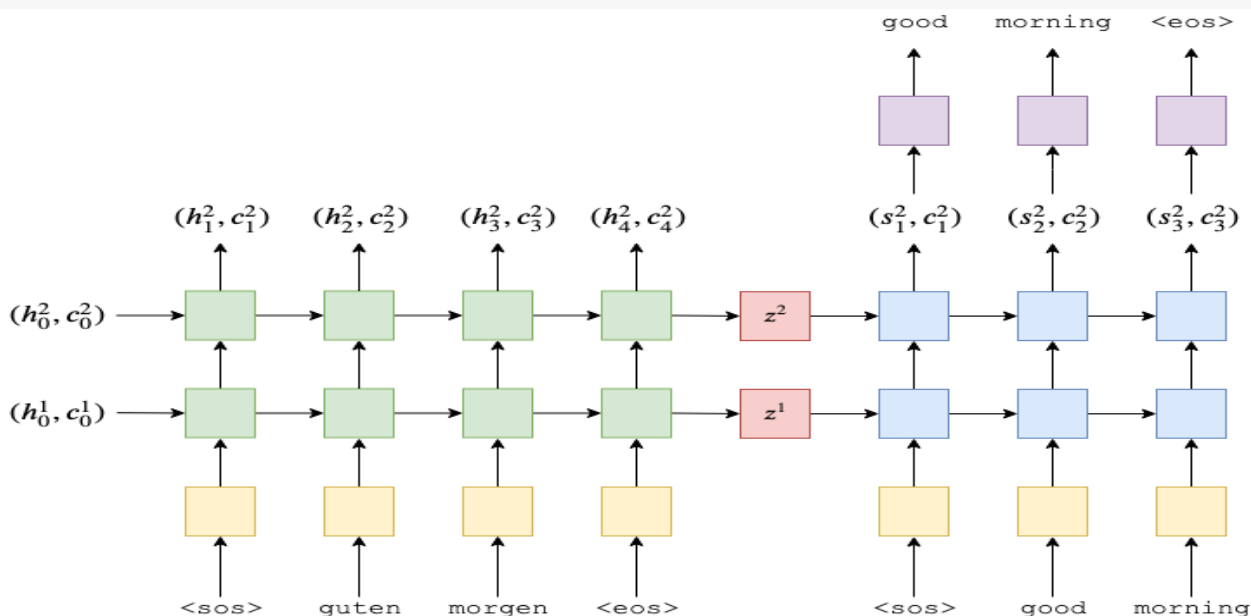
```

Seq2Seq

For the final part of the implementation, we'll implement the seq2seq model. This will handle:

- receiving the input/source sentence
- using the encoder to produce the context vectors
- using the decoder to produce the predicted output/target sentence

Our full model will look like this:



The Seq2Seq model takes in an Encoder, Decoder, and a device (used to place tensors on the GPU, if it exists).

For this implementation, we have to ensure that the number of layers and the hidden (and

cell) dimensions are equal in the Encoder and Decoder.

The teacher forcing ratio is used when training our model. When decoding, at each time-step we will predict what the next token in the target sequence will be from the previous tokens decoded, $y^{t+1}=f(s_t)$. With probability equal to the teaching forcing ratio (teacher_forcing_ratio) we will use the actual ground-truth next token in the sequence as the input to the decoder during the next time-step.

We know how long our target sentences should be (trg_length), so we loop that many times. The last token input into the decoder is the one **before** the <eos> token -- the <eos> token is never input into the decoder.

During each iteration of the loop, we:

- pass the input, previous hidden and previous cell states (y_t, s_{t-1}, c_{t-1}) into the decoder
- receive a prediction, next hidden state and next cell state (y^{t+1}, s_t, c_t) from the decoder
- place our prediction, y^{t+1} /output in our tensor of predictions, Y^t /outputs
- decide if we are going to "teacher force" or not
 - if we do, the next input is the ground-truth next token in the sequence, $y_{t+1}/trg[t]$
 - if we don't, the next input is the predicted next token in the sequence, $y^{t+1}/top1$, which we get by doing an argmax over the output tensor

Once we've made all of our predictions, we return our tensor full of predictions, Y^t /outputs.

$$trg=[< sos >, outputs=[0, y_1, y_2, y_3, < eos >] y^1, y^2, y^3, < eos >]$$

Later on when we calculate the loss, we cut off the first element of each tensor to get:

$$trg=[outputs=[y_1, y_2, y_3, < eos >] y^1, y^2, y^3, < eos >]$$

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device
        assert (
            encoder.hidden_dim == decoder.hidden_dim
        ), "Hidden dimensions of encoder and decoder must be equal!"
        assert (
            encoder.n_layers == decoder.n_layers
        ), "Encoder and decoder must have equal number of layers!"

    def forward(self, src, trg, teacher_forcing_ratio):
        # src = [src length, batch size]
```

```

# trg = [trg length, batch size]
# teacher_forcing_ratio is probability to use teacher forcing
# e.g. if teacher_forcing_ratio is 0.75 we use ground-truth inputs 75% of the time
batch_size = trg.shape[1]
trg_length = trg.shape[0]
trg_vocab_size = self.decoder.output_dim
# tensor to store decoder outputs
outputs = torch.zeros(trg_length, batch_size, trg_vocab_size).to(self.device)
# last hidden state of the encoder is used as the initial hidden state of the
decoder
hidden, cell = self.encoder(src)
# hidden = [n layers * n directions, batch size, hidden dim]
# cell = [n layers * n directions, batch size, hidden dim]
# first input to the decoder is the <sos> tokens
input = trg[0, :]
# input = [batch size]
for t in range(1, trg_length):
    # insert input token embedding, previous hidden and previous cell states
    # receive output tensor (predictions) and new hidden and cell states
    output, hidden, cell = self.decoder(input, hidden, cell)
    # output = [batch size, output dim]
    # hidden = [n layers, batch size, hidden dim]
    # cell = [n layers, batch size, hidden dim]
    # place predictions in a tensor holding predictions for each token
    outputs[t] = output
    # decide if we are going to use teacher forcing or not
    teacher_force = random.random() < teacher_forcing_ratio
    # get the highest predicted token from our predictions
    top1 = output.argmax(1)
# if teacher forcing, use actual next token as next input
# if not, use predicted token
input = trg[t] if teacher_force else top1
# input = [batch size]
return outputs

```

Training the Model

NOW we have our model implemented, we can begin training it.

Model Initialization

We then define the encoder, decoder and then our Seq2Seq model, which we place on the device. The device is used to tell PyTorch whether a model or a tensor should be processed on a GPU or CPU. The `torch.cuda.is_available()` function returns True if a GPU is detected on our machine. Thus, our model will be placed on the GPU, if we have one.

```
input_dim = len(de_vocab)
```

```

output_dim = len(en_vocab)
encoder_embedding_dim = 256
decoder_embedding_dim = 256
hidden_dim = 512
n_layers = 2
encoder_dropout = 0.5
decoder_dropout = 0.5
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
encoder = Encoder(
    input_dim,
    encoder_embedding_dim,
    hidden_dim,
    n_layers,
    encoder_dropout,
)
decoder = Decoder(
    output_dim,
    decoder_embedding_dim,
    hidden_dim,
    n_layers,
    decoder_dropout,
)
model = Seq2Seq(encoder, decoder, device).to(device)

```

Weight Initialization

Next up is initializing the weights of our model. In the paper they state they initialize all weights from a uniform distribution between -0.08 and +0.08, i.e. $U(-0.08, 0.08)$.

We initialize weights in PyTorch by creating a function which we apply to our model. When using apply, the `init_weights` function will be called on every module and sub-module within our model. For each module we loop through all of the parameters and sample them from a uniform distribution with `nn.init.uniform_`.

```

def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.uniform_(param.data, -0.08, 0.08)

```

```

model.apply(init_weights)
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(7853, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.5)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(5893, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.5)
    (fc_out): Linear(in_features=512, out_features=5893, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)

```

```

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f"The model has {count_parameters(model):,} trainable parameters")

```

The model has 13,898,501 trainable parameters

Optimizer

We define our optimizer, which we use to update our parameters in the training loop.

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

Loss Function

The CrossEntropyLoss function calculates both the log softmax as well as the negative log-likelihood of our predictions.

loss function calculates the average loss per token, however by passing the index of the <pad> token as the ignore_index argument we ignore the loss whenever the target token is a padding token.

```
criterion = nn.CrossEntropyLoss(ignore_index=pad_index)
```

Training Loop

we'll set the model into "training mode" with `model.train()`. This will turn on dropout (and batch normalization, which we aren't using) and then iterate through our data iterator.

This means the 0th element of our outputs tensor remains all zeros. So our trg and outputs look something like:

```
trg=[<sos>,outputs=[0,y1,y2,y3,<eos>]y^1,y^2,y^3,<eos>]
```

Here, when we calculate the loss, we cut off the first element of each tensor to get:

```
trg=[outputs=[y1,y2,y3,<eos>]y^1,y^2,y^3,<eos>]
```

At each iteration:

- get the source and target sentences from the batch, X and Y
- zero the gradients calculated from the last batch
- feed the source and target into the model to get the output, Y^
- as the loss function only works on 2d inputs with 1d targets we need to flatten each of them with .view
 - we slice off the first column of the output and target tensors as mentioned above
- calculate the gradients with loss.backward()
- clip the gradients to prevent them from exploding (a common issue in RNNs)
- update the parameters of our model by doing an optimizer step
- sum the loss value to a running total

```
def train_fn(
    model, data_loader, optimizer, criterion, clip, teacher_forcing_ratio, device
):
    model.train()
    epoch_loss = 0
    for i, batch in enumerate(data_loader):
        src = batch["de_ids"].to(device)
        trg = batch["en_ids"].to(device)
        # src = [src length, batch size]
        # trg = [trg length, batch size]
        optimizer.zero_grad()
        output = model(src, trg, teacher_forcing_ratio)
        # output = [trg length, batch size, trg vocab size]
        output_dim = output.shape[-1]
        output = output[1:].view(-1, output_dim)
        # output = [(trg length - 1) * batch size, trg vocab size]
        trg = trg[1:].view(-1)
        # trg = [(trg length - 1) * batch size]
        loss = criterion(output, trg)
        loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
    optimizer.step()
    epoch_loss += loss.item()
    return epoch_loss / len(data_loader)
```

Evaluation Loop

To evaluation mode with `model.eval()`. This will turn off dropout (and batch normalization, if used).

We use the `with torch.no_grad()` block to ensure no gradients are calculated within the block. This reduces memory consumption and speeds things up.

```
def evaluate_fn(model, data_loader, criterion, device):
    model.eval()
    epoch_loss = 0
    with torch.no_grad():
        for i, batch in enumerate(data_loader):
            src = batch["de_ids"].to(device)
            trg = batch["en_ids"].to(device)
            # src = [src length, batch size]
            # trg = [trg length, batch size]
            output = model(src, trg, 0) # turn off teacher forcing
            # output = [trg length, batch size, trg vocab size]
            output_dim = output.shape[-1]
            output = output[1:].view(-1, output_dim)
            # output = [(trg length - 1) * batch size, trg vocab size]
            trg = trg[1:].view(-1)
            # trg = [(trg length - 1) * batch size]
            loss = criterion(output, trg)
            epoch_loss += loss.item()
    return epoch_loss / len(data_loader)
```

Model Training

If it has, we'll update our best validation loss and save the parameters of our model (called `state_dict` in PyTorch). Then, when we come to test our model

```
n_epochs = 10
clip = 1.0
teacher_forcing_ratio = 0.5

best_valid_loss = float("inf")

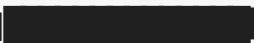
for epoch in tqdm.tqdm(range(n_epochs)):
    train_loss = train_fn(
        model,
        train_data_loader,
        optimizer,
        criterion,
        clip,
        teacher_forcing_ratio,
```



```


        device,
    )
    valid_loss = evaluate_fn(
model,
        valid_data_loader,
        criterion,
        device,
    )
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), "tut1-model.pt")
    print(f"\tTrain Loss: {train_loss:7.3f} | Train PPL: {np.exp(train_loss):7.3f}")
    print(f"\tValid Loss: {valid_loss:7.3f} | Valid PPL: {np.exp(valid_loss):7.3f}")

```

10%  | 1/10
[00:15<02:21, 15.76s/it]


Train Loss: 5.035 | Train PPL: 153.632

Valid Loss: 4.991 | Valid PPL: 147.028

20% 
| 2/10 [00:31<02:05, 15.71s/it]


Train Loss: 4.399 | Train PPL: 81.330

Valid Loss: 4.661 | Valid PPL: 105.741

30% 
| 3/10 [00:47<01:50, 15.73s/it]


Train Loss: 4.075 | Train PPL: 58.860

Valid Loss: 4.487 | Valid PPL: 88.884

40% 
| 4/10 [01:02<01:33, 15.62s/it]

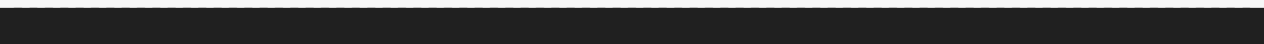
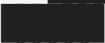
Train Loss: 3.873 | Train PPL: 48.071

Valid Loss: 4.274 | Valid PPL: 71.819

50% 
| 5/10 [01:18<01:18, 15.63s/it]

Train Loss: 3.688 | Train PPL: 39.972

Valid Loss: 4.209 | Valid PPL: 67.278

60% 
 | 6/10 [01:33<01:02, 15.64s/it]

Train Loss: 3.541 | Train PPL: 34.492

Valid Loss: 4.075 | Valid PPL: 58.835

70% 

| 7/10 [01:49<00:46, 15.65s/it]

Train Loss: 3.404 | Train PPL: 30.074

Valid Loss: 4.034 | Valid PPL: 56.458

80%

| 8/10 [02:05<00:31, 15.68s/it]

Train Loss: 3.248 | Train PPL: 25.751

Valid Loss: 3.987 | Valid PPL: 53.888

90%

| 9/10 [02:20<00:15, 15.65s/it]

Train Loss: 3.109 | Train PPL: 22.396

Valid Loss: 3.894 | Valid PPL: 49.093

100%

| 10/10 [02:36<00:00, 15.66s/it]

Train Loss: 2.986 | Train PPL: 19.802

Valid Loss: 3.787 | Valid PPL: 44.137

We've now successfully trained a model that translates German into English! But how well does it perform?

keyboard_arrow_down

Evaluating the Model

The first thing to do is to test the model's performance on the test set.

We'll load the parameters (state_dict) that gave our model the best validation loss and run it on the test set to get our test loss and perplexity.

```
model.load_state_dict(torch.load("tut1-model.pt"))
```

```
test_loss = evaluate_fn(model, test_data_loader, criterion, device)
```

```
print(f"| Test Loss: {test_loss:.3f} | Test PPL: {np.exp(test_loss):7.3f} |")
```

account_circle

| Test Loss: 3.780 | Test PPL: 43.833 |

Pretty similar to the validation performance, which is a good sign. It means we aren't overfitting on the validation set.

You might think it's impossible to overfit on the validation set, but it's not. Every time you tweak your hyperparameters (e.g. optimizer, learning rate, model architecture, weight initialization, etc.) in order to get better results on the validation set, you are slowly overfitting those hyperparameters to your validation set. You can also do this on the test set too! Hence, you should evaluate your model on your test set as few times as possible.

Most papers using neural networks for translation don't give their results in terms of loss and perplexity on the test set, they usually give the [BLEU](#) score. Unlike loss/perplexity, BLEU is a value between zero and one, where higher is better, and [according to the original BLEU paper](#) it has a high correlation with human judgement.

To get our model's BLEU score on the test set, we first need to use our model to translate every example from our test set, which we do with the `translate_sentence` function below. The function first converts the input sentence into tokens, optionally lowercases each token, and then appends the start and end of sequence tokens, `sos_token` and `eos_token`. It then uses the vocabulary to numericalize the tokens into ids and converts these ids into a tensor, adding a "fake" batch dimension, and then passes the tensor through the encoder to get the hidden and cell states. We then perform the decoding, starting with the `sos_token`, converting it into a tensor, passing it through the decoder, getting the `predicted_token` our model thinks is most likely to be next in the sequence, which we append to our list of inputs to the decoder. If the `predicted_token` is the end of sequence token then we stop decoding, if not we continue the loop, using the `predicted_token` as the next input to the decoder. We keep decoding until the decoder outputs the `eos_token` or we hit `max_output_length` (which we use to avoid the decoder just generating tokens forever). Once we've stopped decoding, we convert our inputs into tokens using our vocabulary and return them.

```
def translate_sentence(
    sentence,
    model,
    en_nlp,
    de_nlp,
    en_vocab,
    de_vocab,
    lower,
    sos_token,
    eos_token,
    device,
    max_output_length=25,
```

```

):
    model.eval()
    with torch.no_grad():
        if isinstance(sentence, str):
            tokens = [token.text for token in de_nlp.tokenizer(sentence)]
        else:
            tokens = [token for token in sentence]
        if lower:
            tokens = [token.lower() for token in tokens]
        tokens = [sos_token] + tokens + [eos_token]
        ids = de_vocab.lookup_indices(tokens)
        tensor = torch.LongTensor(ids).unsqueeze(-1).to(device)
        hidden, cell = model.encoder(tensor)
        inputs = en_vocab.lookup_indices([sos_token])
        for _ in range(max_output_length):
            inputs_tensor = torch.LongTensor([inputs[-1]]).to(device)
            output, hidden, cell = model.decoder(inputs_tensor, hidden, cell)
            predicted_token = output.argmax(-1).item()
            inputs.append(predicted_token)
            if predicted_token == en_vocab[eos_token]:
                break
        tokens = en_vocab.lookup_tokens(inputs)
    return tokens

```

We'll pass a test example (something the model hasn't been trained on) to use as a sentence to test our `translate_sentence` function, passing in the German sentence and expecting to get something that looks like the English sentence.

```

sentence = test_data[0]["de"]
expected_translation = test_data[0]["en"]

sentence, expected_translation
account_circle
('Ein Mann mit einem orangefarbenen Hut, der etwas anstarrt.',
'A man in an orange hat starring at something.')

```

```

translation = translate_sentence(
    sentence,
    model,
    en_nlp,
    de_nlp,
    en_vocab,

```

```
de_vocab,  
lower,  
    sos_token,  
eos_token,  
device,  
)
```

Our model has seemed to have figured out that the input sentence mentions a man wearing an item of clothing (though gets both the color and the item wrong), but it can't seem to figure out what the man is doing.

We shouldn't be expecting amazing results, our model is relatively small to what is used in the paper we're implementing (they use four layers with embedding and hidden dimensions of 1000) and is miniscule compared to modern translation models (which have billions of parameters).

translation

account_circle

```
['<sos>',  
'a',  
'man',  
'in',  
'a',  
'white',  
'shirt',  
'is',  
'cutting',  
'something',  
'.',  
'<eos>']
```

The model doesn't just translate examples in the training, validation and test sets. We can use it to translate arbitrary sentences by passing any string to the `translate_sentence`.

Note that the multi30k dataset consists of image captions that have been translated from English to German, and our model has been trained to translate German to English. Therefore, the model will only output reasonable translations if the sentences are German

sentences that could potentially be image captions. (It's also important to re-iterate that the model trained here is relatively small and the translation performance will generally be poor.)

Below, we input the German translation of **"A man is watching a film."**

```
sentence = "Ein Mann sitzt auf einer Bank."
```

```
translation = translate_sentence(
    sentence,
    model,
    en_nlp,
    de_nlp,
    en_vocab,
    de_vocab,
    lower,
    sos_token,
    eos_token,
    device,
)
```

And we receive our translation, which is reasonably close.

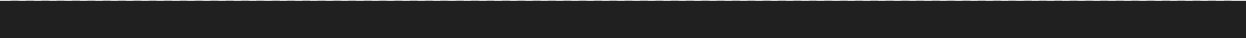
```
translation
account_circle
['<sos>', 'man', 'sitting', 'on', 'a', 'bench', '.', '<eos>']
```

We can now loop over our `test_data`, getting our model's translation of each test sentence.

```
translations = [
    translate_sentence(
        example["de"],
        model,
        en_nlp,
        de_nlp,
        en_vocab,
        de_vocab,
        lower,
        sos_token,
        eos_token,
```

```
device,
)
for example in tqdm.tqdm(test_data)
]
```

account_circle

100%  1000/1000 [00:04<00:00, 235.22it/s]

To calculate BLEU, we'll use the evaluate library. It's recommended to use libraries for measuring metrics to ensure there are no bugs in your metric calculations and giving you potentially incorrect results.

The BLEU metric can be loaded from the evaluate library like so:

```
bleu = evaluate.load("bleu")
```

One quirk one the BLEU metric is that it expects the predictions (predicted translations) to be strings and the references (actual English sentences) to be a list of sentences. This is because BLEU works if you have multiple correct sentences per prediction as there may be potentially be multiple ways to translate a sentence. In our case, we only have a single reference sentence so we just wrap our target sentence in a list. We also convert our translations from a list of tokens into a string by joining them with whitespace inbetween and getting rid of the <eos> and <eos> tokens (as they will never appear in our reference sentences).

```
predictions = [" ".join(translation[1:-1]) for translation in translations]
```

```
references = [[example["en"]] for example in test_data]
```

```
predictions[0], references[0]
```

account_circle

```
('a man in a white shirt is cutting something .',
```

```
['A man in an orange hat starring at something.'])
```

We also need to define a function which tokenizes an input string. This will be used to calculate the BLEU score by comparing our predicted tokens against the reference tokens.

It seems a bit odd that we joined our translated tokens together into a string only to just tokenize them

again, and also used the English string from our test data instead of the existing tokens (en_tokens), however this is another quirk of the BLEU metric provided by the evaluate library; the predictions and references must be strings and not tokens, and that we must tell the metric how these strings should be tokenized.

The get_tokenize_fn returns our tokenizer_fn, which uses our spaCy tokenizer and lowercases tokens if necessary.

```
def get_tokenizer_fn(nlp, lower):
    def tokenizer_fn(s):
        tokens = [token.text for token in nlp.tokenizer(s)]
        if lower:
            tokens = [token.lower() for token in tokens]
        return tokens

    return tokenizer_fn
```

```
tokenizer_fn = get_tokenizer_fn(en_nlp, lower)
```

```
tokenizer_fn(predictions[0]), tokenizer_fn(references[0][0])
account_circle
(['a', 'man', 'in', 'a', 'white', 'shirt', 'is', 'cutting', 'something', '.'],
 ['a', 'man', 'in', 'an', 'orange', 'hat', 'starring', 'at', 'something', '.'])
```

Finally, we calculate the BLEU metric across our test set!

We pass our predictions, references and our tokenizer_fn to the compute method of the BLEU metric to get our results.

```
results = bleu.compute(
    predictions=predictions, references=references, tokenizer=tokenizer_fn
)
```

Results:

```
account_circle
{'bleu': 0.13756994726803526,
 'precisions': [0.4681576952236543,
 0.18843314191960622,
 0.09133154602323502,
```



```
0.04445534838076546],  
'brevity_penalty': 1.0,  
'length_ratio': 1.0101087455965692,  
'translation_length': 13190,  
'reference_length': 13058}
```

We get a BLEU score of 0.14! Nothing to write home about, but not bad for our first translation model.

In the subsequent notebooks we'll be implementing more translation papers and slowly increasing the BLEU score achieved.
