Introduction

In this notebook we will be adding a few improvements - packed padded sequences and masking - to the model from the previous notebook. Packed padded sequences are used to tell our RNN to skip over padding tokens in our encoder. Masking explicitly forces the model to ignore certain values, such as attention over padded elements. Both of these techniques are commonly used in NLP.

We will also look at how to use our model for inference, by giving it a sentence, seeing what it translates it as and seeing where exactly it pays attention to when translating each word.

Finally, we'll use the BLEU metric to measure the quality of our translations.

Preparing Data

First, we'll import all the modules as before, with the addition of the `matplotlib` modules used for viewing the attention.

In [1]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from torchtext.legacy.datasets import Multi30k
from torchtext.legacy.data import Field, BucketIterator, TabularDataset

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

import spacy
import numpy as np

import random
import math
import time
import json
```

Next, we'll set the random seed for reproducability.

In [2]:
```python
SEED = 1234

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

As before, we'll import spaCy and define the German and English tokenizers.

In [3]:
```python
spacy_de = spacy.load('de_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')
```

In [4]:
```python
def tokenize_de(text):
    """
    Tokenizes German text from a string into a list of strings
    """
    return [tok.text for tok in spacy_de.tokenizer(text)]

def tokenize_en(text):
    """
    Tokenizes English text from a string into a list of strings
    """
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

When using packed padded sequences, we need to tell PyTorch how long the actual (non-padded) sequences are. Luckily for us, TorchText's `Field` objects allow us to use the `include_lengths` argument, this will cause our `batch.src` to be a tuple. The first element of the tuple is the same as before, a batch of numericalized source sentence as a tensor, and the second element is the non-padded lengths of each source sentence within the batch.

In [5]:

```
SRC = Field(
#   tokenize = tokenize_de,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True,
    include_lengths = True
)

TRG = Field(
    #tokenize = tokenize_en,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True
)
```

We then load the data.

In [6]:

```
# NOTE: this line takes a long time to run on Colab so
# instead we'll load the already tokenized json dataset
# from Github
#
# train_data, valid_data, test_data = Multi30k.splits(exts = ('.de', '.en'),
#                                                      fields = (SRC, TRG))

# fetch from Github repo
!wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/train.json
!wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/valid.json
!wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/test.json

# and load to same variables
fields = {'src': ('src', SRC), 'trg': ('trg', TRG)}
train_data, valid_data, test_data = TabularDataset.splits(
    path = '.',
    train = 'train.json',
    validation = 'valid.json',
    test = 'test.json',
    format = 'json',
    fields = fields
)
```

```
'wget' is not recognized as an internal or external command,
operable program or batch file.
'wget' is not recognized as an internal or external command,
operable program or batch file.
'wget' is not recognized as an internal or external command,
operable program or batch file.
```

And build the vocabulary.

In [7]:
```
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)
```

Next, we handle the iterators.

One quirk about packed padded sequences is that all elements in the batch need to be sorted by their non-padded lengths in descending order, i.e. the first sentence in the batch needs to be the longest. We use two arguments of the iterator to handle this, `sort_within_batch` which tells the iterator that the contents of the batch need to be sorted, and `sort_key` a function which tells the iterator how to sort the elements in the batch. Here, we sort by the length of the `src` sentence.

In [8]:
```
BATCH_SIZE = 128

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
     batch_size = BATCH_SIZE,
     sort_within_batch = True,
     sort_key = lambda x : len(x.src),
     device = device)
```

Building the Model

Encoder

Next up, we define the encoder.

The changes here all within the `forward` method. It now accepts the lengths of the source sentences as well as the sentences themselves.

After the source sentence (padded automatically within the iterator) has been embedded, we can then use `pack_padded_sequence` on it with the lengths of the sentences. Note that the tensor containing the lengths of the sequences must be a CPU tensor as of the latest version of PyTorch, which we explicitly do so with `to('cpu')`. `packed_embedded` will then be our packed padded sequence. This can be then fed to our RNN as normal which will return `packed_outputs`, a packed tensor containing all of the hidden states from the sequence, and `hidden` which is simply the final hidden state from our sequence. `hidden` is a standard tensor and not packed in any way, the only difference is that as the input was a packed sequence, this tensor is from the final **non-padded element** in the sequence.

We then unpack our `packed_outputs` using `pad_packed_sequence` which returns the `outputs` and the lengths of each, which we don't need.

The first dimension of `outputs` is the padded sequence lengths however due to using a packed padded sequence the values of tensors when a padding token was the input will be all zeros.

In [9]:
```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_len):

        #src = [src len, batch size]
        #src_len = [batch size]

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

        #embedded = [src len, batch size, emb dim]

        #need to explicitly put lengths on cpu!
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, src_len.to('cpu'))

        packed_outputs, hidden = self.rnn(packed_embedded)

        #packed_outputs is a packed sequence containing all hidden states
        #hidden is now from the final non-padded element in the batch

        outputs, _ = nn.utils.rnn.pad_packed_sequence(packed_outputs)

        #outputs is now a non-packed sequence, all hidden states obtained
        #  when the input is a pad token are all zeros

        #outputs = [src len, batch size, hid dim * num directions]
        #hidden = [n layers * num directions, batch size, hid dim]

        #hidden is stacked [forward_1, backward_1, forward_2, backward_2, ...]
        #outputs are always from the last layer

        #hidden [-2, :, : ] is the last of the forwards RNN
        #hidden [-1, :, : ] is the last of the backwards RNN

        #initial decoder hidden is final hidden state of the forwards and backwards
        #  encoder RNNs fed through a linear layer
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)))

        #outputs = [src len, batch size, enc hid dim * 2]
        #hidden = [batch size, dec hid dim]

        return outputs, hidden
```

Attention

The attention module is where we calculate the attention values over the source sentence.

Previously, we allowed this module to "pay attention" to padding tokens within the source sentence. However, using *masking*, we can force the attention to only be over non-padding elements.

The `forward` method now takes a `mask` input. This is a **[batch size, source sentence length]** tensor that is 1 when the source sentence token is not a padding token, and 0 when it is a padding token. For example, if the source sentence is: ["hello", "how", "are", "you", "?", `<pad>`, `<pad>`], then the mask would be [1, 1, 1, 1, 1, 0, 0].

We apply the mask after the attention has been calculated, but before it has been normalized by the `softmax` function. It is applied using `masked_fill`. This fills the tensor at each element where the first argument (`mask == 0`) is true, with the value given by the second argument (`-1e10`). In other words, it will take the unnormalized attention values, and change the attention values over padded elements to be `-1e10`. As these numbers will be miniscule compared to the other values they will become zero when passed through the `softmax` layer, ensuring no attention is payed to padding tokens in the source sentence.

In [10]:
```
class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()

        self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim, dec_hid_dim)
        self.v = nn.Linear(dec_hid_dim, 1, bias = False)

    def forward(self, hidden, encoder_outputs, mask):

        #hidden = [batch size, dec hid dim]
        #encoder_outputs = [src len, batch size, enc hid dim * 2]

        batch_size = encoder_outputs.shape[1]
        src_len = encoder_outputs.shape[0]

        #repeat decoder hidden state src_len times
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)

        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        #hidden = [batch size, src len, dec hid dim]
        #encoder_outputs = [batch size, src len, enc hid dim * 2]

        energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim = 2)))

        #energy = [batch size, src len, dec hid dim]

        attention = self.v(energy).squeeze(2)

        #attention = [batch size, src len]

        attention = attention.masked_fill(mask == 0, -1e10)

        return F.softmax(attention, dim = 1)
```

Decoder

The decoder only needs a few small changes. It needs to accept a mask over the source sentence and pass this to the attention module. As we want to view the values of attention during inference, we also return the attention tensor.

In [11]:
```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout, attention):
        super().__init__()

        self.output_dim = output_dim
        self.attention = attention

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.GRU((enc_hid_dim * 2) + emb_dim, dec_hid_dim)

        self.fc_out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs, mask):
        #input = [batch size]
        #hidden = [batch size, dec hid dim]
        #encoder_outputs = [src len, batch size, enc hid dim * 2]
        #mask = [batch size, src len]

        input = input.unsqueeze(0)

        #input = [1, batch size]

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

        #embedded = [1, batch size, emb dim]

        a = self.attention(hidden, encoder_outputs, mask)

        #a = [batch size, src len]

        a = a.unsqueeze(1)

        #a = [batch size, 1, src len]

        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        #encoder_outputs = [batch size, src len, enc hid dim * 2]

        weighted = torch.bmm(a, encoder_outputs)

        #weighted = [batch size, 1, enc hid dim * 2]

        weighted = weighted.permute(1, 0, 2)

        #weighted = [1, batch size, enc hid dim * 2]

        rnn_input = torch.cat((embedded, weighted), dim = 2)

        #rnn_input = [1, batch size, (enc hid dim * 2) + emb dim]

        output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))

        #output = [seq len, batch size, dec hid dim * n directions]
        #hidden = [n layers * n directions, batch size, dec hid dim]

        #seq len, n layers and n directions will always be 1 in this decoder, therefore:
        #output = [1, batch size, dec hid dim]
        #hidden = [1, batch size, dec hid dim]
        #this also means that output == hidden
        assert (output == hidden).all()

        embedded = embedded.squeeze(0)
        output = output.squeeze(0)
        weighted = weighted.squeeze(0)

        prediction = self.fc_out(torch.cat((output, weighted, embedded), dim = 1))

        #prediction = [batch size, output dim]

        return prediction, hidden.squeeze(0), a.squeeze(1)
```

Seq2Seq

The overarching seq2seq model also needs a few changes for packed padded sequences, masking and inference.

We need to tell it what the indexes are for the pad token and also pass the source sentence lengths as input to the `forward` method.

We use the pad token index to create the masks, by creating a mask tensor that is 1 wherever the source sentence is not equal to the pad token. This is all done within the `create_mask` function.

The sequence lengths as needed to pass to the encoder to use packed padded sequences.

The attention at each time-step is stored in the `attentions`

In [12]:
```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, src_pad_idx, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_idx = src_pad_idx
        self.device = device

    def create_mask(self, src):
        mask = (src != self.src_pad_idx).permute(1, 0)
        return mask

    def forward(self, src, src_len, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #src_len = [batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use teacher forcing 75% of the time

        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        #encoder_outputs is all hidden states of the input sequence, back and forwards
        #hidden is the final forward and backward hidden states, passed through a linear
        encoder_outputs, hidden = self.encoder(src, src_len)

        #first input to the decoder is the <sos> tokens
        input = trg[0,:]

        mask = self.create_mask(src)

        #mask = [batch size, src len]

        for t in range(1, trg_len):

            #insert input token embedding, previous hidden state, all encoder hidden state
            #  and mask
            #receive output tensor (predictions) and new hidden state
            output, hidden, _ = self.decoder(input, hidden, encoder_outputs, mask)

            #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)
            #print(output)
            #print(output.shape)
            #print(t)
            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1


        return outputs
```

Training the Seq2Seq Model

Next up, initializing the model and placing it on the GPU.

In [13]:

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
ENC_HID_DIM = 512
DEC_HID_DIM = 512
ENC_DROPOUT = 0.5
DEC_DROPOUT = 0.5
SRC_PAD_IDX = SRC.vocab.stoi[SRC.pad_token]

attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT, attn)

model = Seq2Seq(enc, dec, SRC_PAD_IDX, device).to(device)
```

Then, we initialize the model parameters.

In [14]:

```
def init_weights(m):
    for name, param in m.named_parameters():
        if 'weight' in name:
            nn.init.normal_(param.data, mean=0, std=0.01)
        else:
            nn.init.constant_(param.data, 0)

model.apply(init_weights)
```

Out[14]:

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(7853, 256)
    (rnn): GRU(256, 512, bidirectional=True)
    (fc): Linear(in_features=1024, out_features=512, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (attention): Attention(
      (attn): Linear(in_features=1536, out_features=512, bias=True)
      (v): Linear(in_features=512, out_features=1, bias=False)
    )
    (embedding): Embedding(5893, 256)
    (rnn): GRU(1280, 512)
    (fc_out): Linear(in_features=1792, out_features=5893, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

We'll print out the number of trainable parameters in the model, noticing that it has the exact same amount of parameters as the model without these improvements.

In [15]:
```python
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 20,518,405 trainable parameters
```

Then we define our optimizer and criterion.

The `ignore_index` for the criterion needs to be the index of the pad token for the target language, not the source language.

In [16]:
```python
optimizer = optim.Adam(model.parameters())
```

In [17]:
```python
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

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

Next, we'll define our training and evaluation loops.

As we are using `include_lengths = True` for our source field, `batch.src` is now a tuple with the first element being the numericalized tensor representing the sentence and the second element being the lengths of each sentence within the batch.

Our model also returns the attention vectors over the batch of source source sentences for each decoding time-step. We won't use these during the training/evaluation, but we will later for inference.

In [18]:
```python
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in enumerate(iterator):

        src, src_len = batch.src
        trg = batch.trg

        optimizer.zero_grad()

        output = model(src, src_len, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]

        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(iterator)
```

In [19]:

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in enumerate(iterator):

            src, src_len = batch.src
            trg = batch.trg

            output = model(src, src_len, trg, 0) #turn off teacher forcing

            #trg = [trg len, batch size]
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)

            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

Then, we'll define a useful function for timing how long epochs take.

In [20]:

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

The penultimate step is to train our model. Notice how it takes almost half the time as our model without the improvements added in this notebook.

In [21]:

```
import pandas as pd
df = pd.DataFrame()
```

In [22]:

```
N_EPOCHS = 10
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut4-model.pt')

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
    dict = {'train_loss': train_loss, 'Val_loss': valid_loss, 'epoch': epoch}
    df = df.append(dict, ignore_index = True )
```

```
Epoch: 01 | Time: 0m 27s
	Train Loss: 5.191 | Train PPL: 179.649
	 Val. Loss: 5.071 |  Val. PPL: 159.264
Epoch: 02 | Time: 0m 25s
	Train Loss: 4.492 | Train PPL:  89.318
	 Val. Loss: 4.754 |  Val. PPL: 116.104
Epoch: 03 | Time: 0m 26s
	Train Loss: 3.784 | Train PPL:  43.978
	 Val. Loss: 3.825 |  Val. PPL:  45.853
Epoch: 04 | Time: 0m 26s
	Train Loss: 3.052 | Train PPL:  21.160
	 Val. Loss: 3.436 |  Val. PPL:  31.048
Epoch: 05 | Time: 0m 25s
	Train Loss: 2.581 | Train PPL:  13.207
	 Val. Loss: 3.260 |  Val. PPL:  26.056
Epoch: 06 | Time: 0m 25s
	Train Loss: 2.240 | Train PPL:   9.395
	 Val. Loss: 3.208 |  Val. PPL:  24.718
Epoch: 07 | Time: 0m 26s
	Train Loss: 1.981 | Train PPL:   7.252
	 Val. Loss: 3.148 |  Val. PPL:  23.283
Epoch: 08 | Time: 0m 25s
	Train Loss: 1.775 | Train PPL:   5.902
	 Val. Loss: 3.206 |  Val. PPL:  24.687
Epoch: 09 | Time: 0m 25s
	Train Loss: 1.613 | Train PPL:   5.019
	 Val. Loss: 3.235 |  Val. PPL:  25.397
Epoch: 10 | Time: 0m 26s
	Train Loss: 1.504 | Train PPL:   4.498
	 Val. Loss: 3.276 |  Val. PPL:  26.465
```

Finally, we load the parameters from our best validation loss and get our results on the test set.

We get the improved test perplexity whilst almost being twice as fast!

```
In [23]: df
```

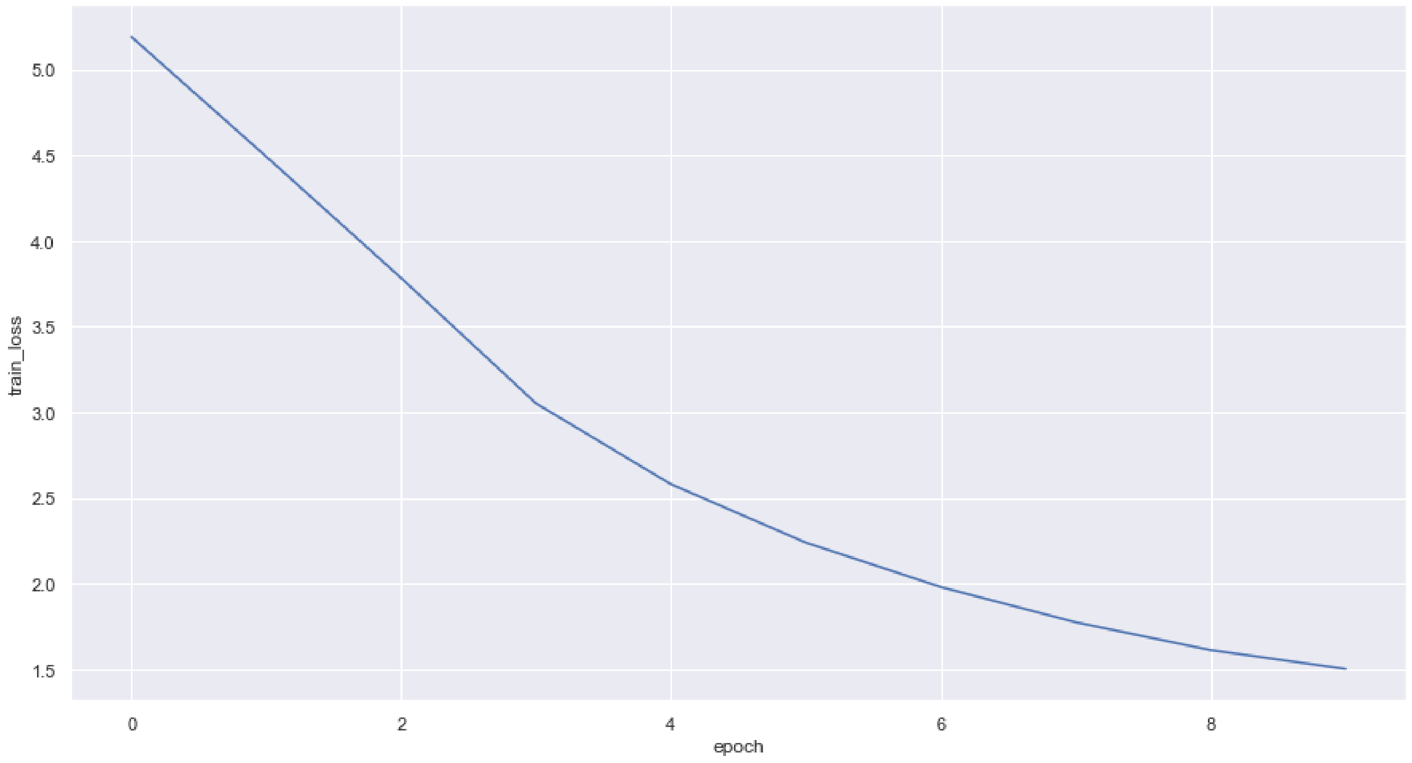
Out[23]:

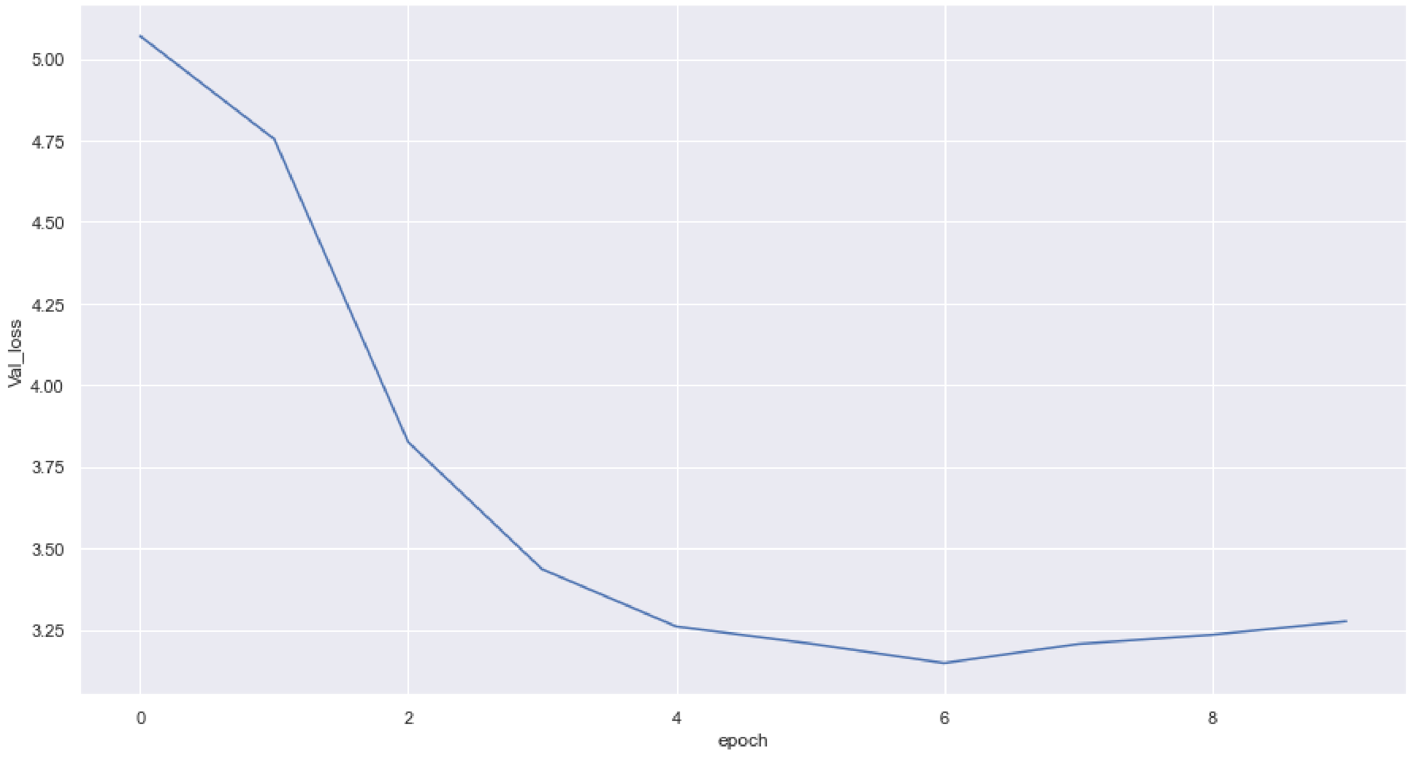
	train_loss	Val_loss	epoch
0	5.191003	5.070560	0.0
1	4.492200	4.754489	1.0
2	3.783691	3.825434	2.0

	train_loss	Val_loss	epoch
3	3.052097	3.435534	3.0
4	2.580736	3.260239	4.0
5	2.240160	3.207526	5.0
6	1.981226	3.147743	6.0
7	1.775329	3.206273	7.0
8	1.613304	3.234632	8.0
9	1.503621	3.275815	9.0

In [24]:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(rc = {'figure.figsize': (15,8)})
sns.lineplot(data = df, x = 'epoch', y = 'train_loss')
plt.show()
sns.lineplot(data = df, x = 'epoch', y = 'Val_loss')
plt.show()
```





In [22]:
```
model.load_state_dict(torch.load('tut4-model.pt'))

test_loss = evaluate(model, test_iterator, criterion)

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

```
| Test Loss: 3.129 | Test PPL:  22.857 |
```

In [23]:
```
import math
```

Inference

Now we can use our trained model to generate translations.

Note: these translations will be poor compared to examples shown in paper as they use hidden dimension sizes of 1000 and train for 4 days! They have been cherry picked in order to show off what attention should look like on a sufficiently sized model.

Our `translate_sentence` will do the following:

- ensure our model is in evaluation mode, which it should always be for inference
- tokenize the source sentence if it has not been tokenized (is a string)
- numericalize the source sentence
- convert it to a tensor and add a batch dimension
- get the length of the source sentence and convert to a tensor
- feed the source sentence into the encoder
- create the mask for the source sentence
- create a list to hold the output sentence, initialized with an `<sos>` token
- create a tensor to hold the attention values
- while we have not hit a maximum length
    - get the input tensor, which should be either `<sos>` or the last predicted token

    - feed the input, all encoder outputs, hidden state and mask into the decoder
    - store attention values
    - get the predicted next token
    - add prediction to current output sentence prediction
    - break if the prediction was an `<eos>` token
- convert the output sentence from indexes to tokens
- return the output sentence (with the `<sos>` token removed) and the attention values over the sequence

In [24]:
```
[(1,2,3,4,5,False)]
```

Out[24]:
```
[(1, 2, 3, 4, 5, False)]
```

In [50]:
```
def translate_sentence(sentence, src_field, trg_field, model, device, max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('de')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]

    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    src_len = torch.LongTensor([len(src_indexes)])

    with torch.no_grad():
        encoder_outputs, hidden = model.encoder(src_tensor, src_len)

    mask = model.create_mask(src_tensor)

    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    attentions = torch.zeros(max_len, 1, len(src_indexes)).to(device)
    #print(type(attentions))
    #print(attentions.shape)

    greedy = False

    beam = False

    m = nn.Softmax(dim = 1)

    if greedy:

        for i in range(max_len):

            trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)

            with torch.no_grad():
                output, hidden, attention = model.decoder(trg_tensor, hidden, encoder_outp

            attentions[i] = attention

            pred_token = output.argmax(1).item()
            trg_indexes.append(pred_token)

            if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
                break
    elif beam:
        alpha = 1
        k = 3
        # need to track top k candidates each time ->
        # ([sequence], probability, length, probability_with_length_penalty, bool: Eos or
        # initialize empty tuple
        sequences = [(trg_indexes, 0.0, 0, 0.0, False, hidden)]
        overall_list = []
        #m = nn.Softmax(dim = 1)
        for row in range(int(len(sentence)*1.4)):
            all_candidates = list()
            # loop each tuple in the seq, k sequences
            for i in range(len(sequences)):
                (seq, score, length, _, eos, curr_hidden) = sequences[i]
                # if this sequence is not done, expand it
                if eos == False:
                    # predict based on the last element of the previous tensor
                    trg_tensor = torch.LongTensor([seq[-1]]).to(device)
                    with torch.no_grad():
                        output, next_hidden, _ = model.decoder(trg_tensor, curr_hidden, \
                                                  encoder_outputs, mask)
                        # softmax the output, now output is a list of probability with pos
                        output = m(output)
                        probability_list, token_index = torch.topk(output, k, dim = 1)
                    for j in range(len(token_index[0])):
                        curr_probability = math.log(probability_list[0][j].item())
                        pred_token = token_index[0][j].item()
                        next_score = score + curr_probability
                        next_penalty = next_score/((length+1)**alpha)
                        # if index of j is eos
                        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
                            next_eos = True
                        else:
                            next_eos = False

                        candidate = (seq + [pred_token], next_score, length + 1, \
                                     next_penalty, next_eos, next_hidden)
                        all_candidates.append(candidate)
                else:
                    # add the eos sequence into the list, no need to expand
                    all_candidates.append(sequences[i])

            ordered = sorted(all_candidates, key = lambda tuples: tuples[3], reverse = Tru
            sequences = ordered[:k]
        trg_indexes = sequences[0][0]



    # Nucleus Search
    else:
        # get the distribution of the current output, which is a list of probabilities (so
        # start with the highest probability, add until exceed p, becomes a subset
        # Rescale
        # Sample a token as the predicted token
        # continue until faced eos

        p = 0.05
        for i in range(max_len):
            trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)
            with torch.no_grad():
                output, hidden, attention = model.decoder(trg_tensor, \
                                                          hidden, encoder_outputs, mask)
            attentions[i] = attention
            # softmax function
            output = m(output)

            probs, index = torch.sort(output, descending = True)
            total = 0.0
            i = 0
            while total < p:
                total += probs[0][i]
                i += 1

            probs[0][i:] = -math.inf

            # softmax the output again
            probs = m(probs)

            pred_index = torch.multinomial(probs, 1, replacement = True).item()
            pred_token = index[0][pred_index]
            trg_indexes.append(pred_token)
            if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
                break

    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]

    return trg_tokens[1:], attentions[:len(trg_tokens)-1]
```

Next, we'll make a function that displays the model's attention over the source sentence for each target token generated.

In [51]:
```
def display_attention(sentence, translation, attention):

    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)

    attention = attention.squeeze(1).cpu().detach().numpy()

    cax = ax.matshow(attention, cmap='bone')

    ax.tick_params(labelsize=15)

    x_ticks = [''] + ['<sos>'] + [t.lower() for t in sentence] + ['<eos>']
    y_ticks = [''] + translation

    ax.set_xticklabels(x_ticks, rotation=45)
    ax.set_yticklabels(y_ticks)

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()
    plt.close()
```

Now, we'll grab some translations from our dataset and see how well our model did. Note, we're going to cherry pick examples here so it gives us something interesting to look at, but feel free to change the `example_idx` value to look at different examples.

First, we'll get a source and target from our dataset.

In [52]:
```
example_idx = 12

#src = vars(train_data.examples[example_idx])['src']
```

```
src1 = ["ein", "angestellter", "reicht", "einer", "frau", "auf", "einem", "markt", "eine",
trg = ["an", "employee", "is", "handing", "a", "woman", "a", "bag", "while", "she", "is",
#print(f'src = {src}')
print(f'src = {src1}')
print(f'trg = {trg}')
```

```
src = ['ein', 'angestellter', 'reicht', 'einer', 'frau', 'auf', 'einem', 'markt', 'eine',
'tüte', ',', 'während', 'sie', 'auf', 'eis', 'gelegten', 'fisch', 'begutachtet', '.']
trg = ['an', 'employee', 'is', 'handing', 'a', 'woman', 'a', 'bag', 'while', 'she', 'is',
'browsing', 'through', 'fish', 'on', 'ice', 'at', 'a', 'street', 'market', '.']
```

Then we'll use our `translate_sentence` function to get our predicted translation and attention. We show this graphically by having the source sentence on the x-axis and the predicted translation on the y-axis. The lighter the square at the intersection between two words, the more attention the model gave to that source word when translating that target word.

Below is an example the model attempted to translate, it gets the translation correct except changes *are fighting* to just *fighting*.

In [53]:

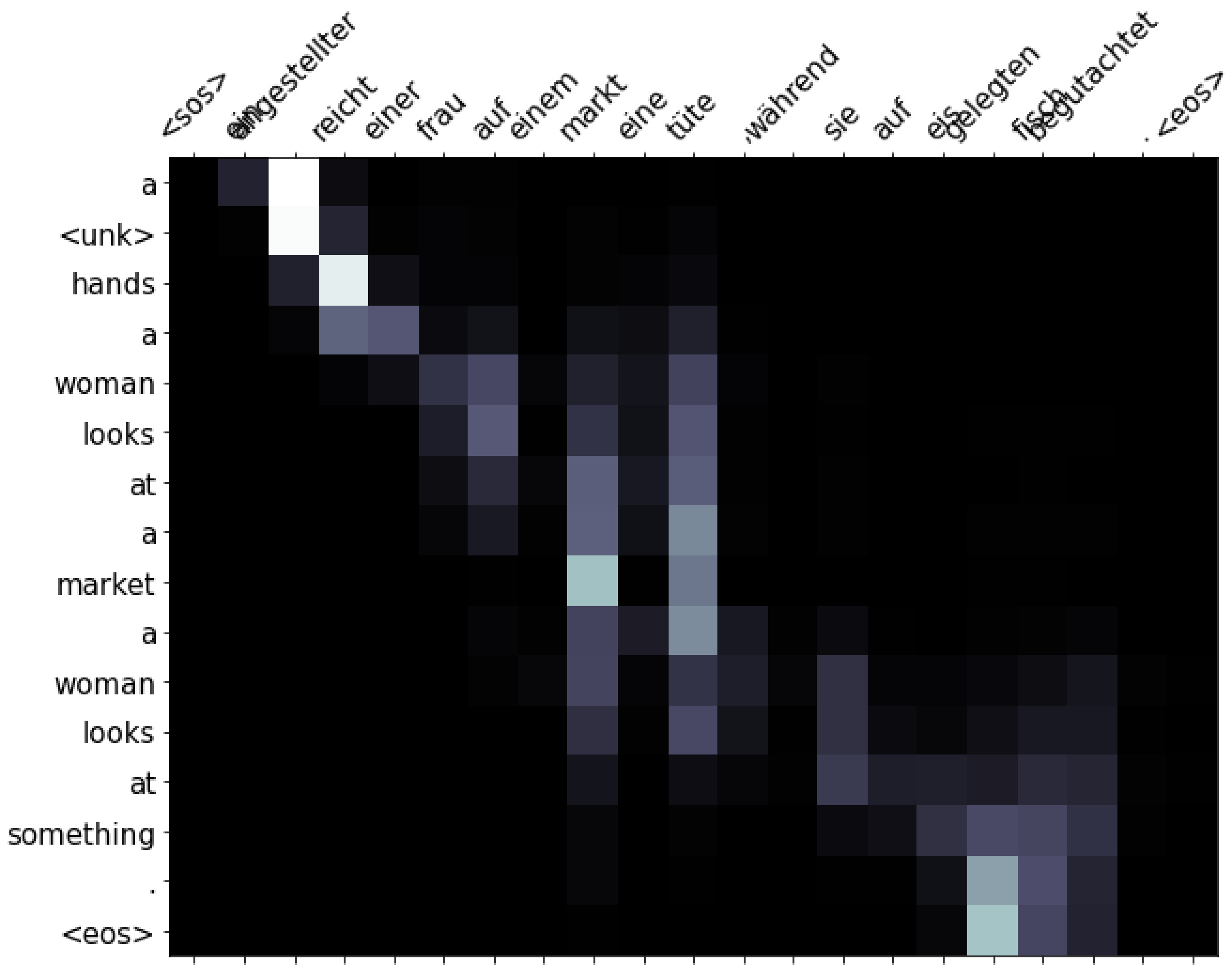
```
translation, attention = translate_sentence(src1, SRC, TRG, model, device)

print(f'predicted trg = {translation}')
```

```
predicted trg = ['a', '<unk>', 'hands', 'a', 'woman', 'looks', 'at', 'a', 'market', 'a',
'woman', 'looks', 'at', 'something', '.', '<eos>']
```

In [54]:

```
display_attention(src1, translation, attention)
```

```
C:\Users\yoush\AppData\Local\Temp/ipykernel_19488/4268658779.py:15: UserWarning: FixedForm
atter should only be used together with FixedLocator
  ax.set_xticklabels(x_ticks, rotation=45)
C:\Users\yoush\AppData\Local\Temp/ipykernel_19488/4268658779.py:16: UserWarning: FixedForm
atter should only be used together with FixedLocator
  ax.set_yticklabels(y_ticks)
```



Translations from the training set could simply be memorized by the model. So it's only fair we look at translations from the validation and testing set too.

Starting with the validation set, let's get an example.

In [40]:

```python
example_idx = 14

src = ["eine", "gruppe", "von", "kindern", "im", "unterricht", ",", "die", "sich", "zu", '
trg = ["a", "group", "of", "kids", "in", "class", "with", "their", "hands", "in", "the", '

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['eine', 'gruppe', 'von', 'kindern', 'im', 'unterricht', ',', 'die', 'sich', 'zu',
'wort', 'melden', '.']
trg = ['a', 'group', 'of', 'kids', 'in', 'class', 'with', 'their', 'hands', 'in', 'the',
'air', '.']
```

Then let's generate our translation and view the attention.

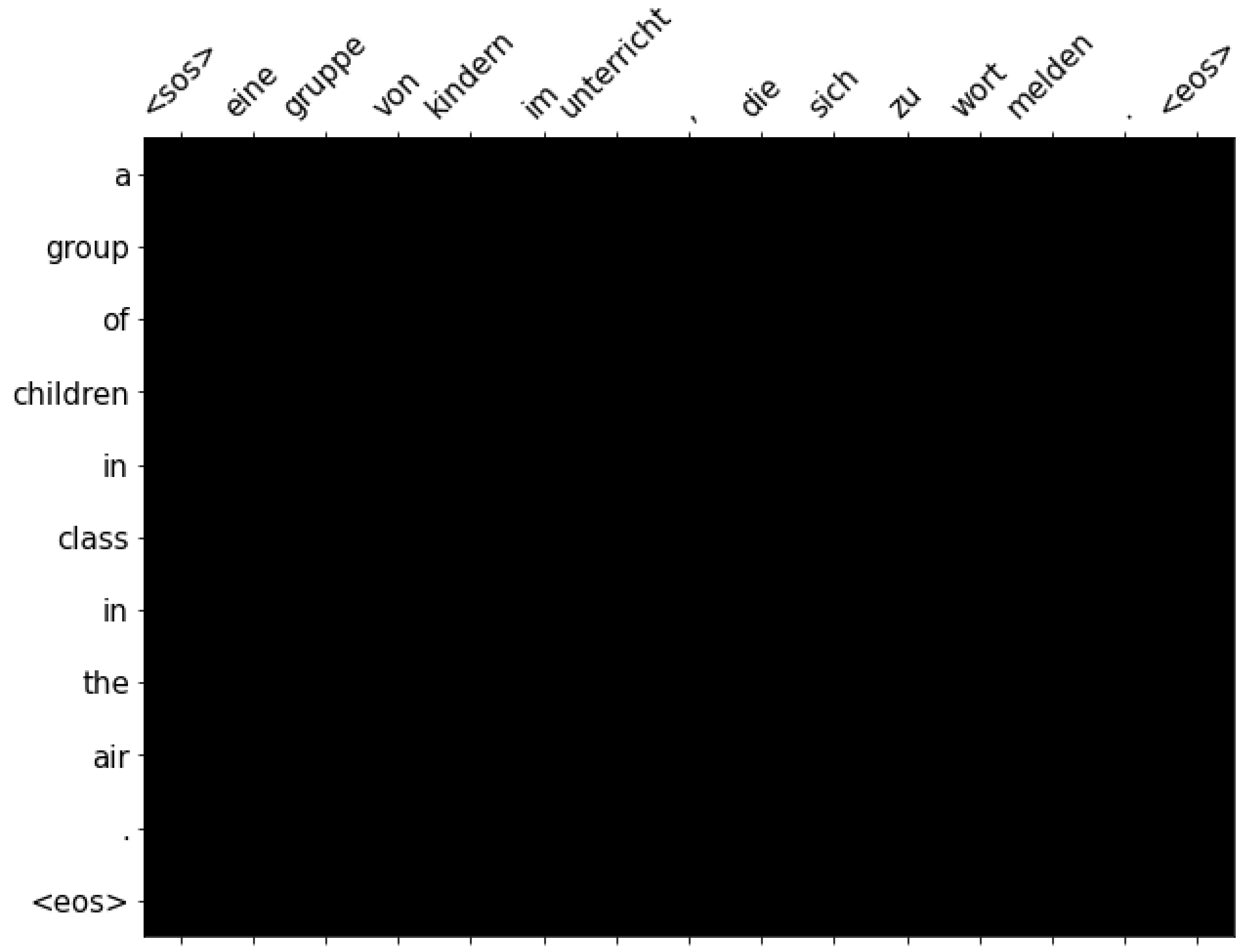
Here, we can see the translation is the same except for swapping *female* with *woman*.

In [41]:

```python
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')

display_attention(src, translation, attention)
```

```
predicted trg = ['a', 'group', 'of', 'children', 'in', 'class', 'in', 'the', 'air', '.',
'<eos>']
```

```
C:\Users\yoush\AppData\Local\Temp/ipykernel_19488/4268658779.py:15: UserWarning: FixedForm
atter should only be used together with FixedLocator
  ax.set_xticklabels(x_ticks, rotation=45)
C:\Users\yoush\AppData\Local\Temp/ipykernel_19488/4268658779.py:16: UserWarning: FixedForm
atter should only be used together with FixedLocator
  ax.set_yticklabels(y_ticks)
```



Finally, let's get an example from the test set.

In [42]:
```python
example_idx = 18

src = ["zwei", "m\u00e4nner", ",", "die", "am", "strand", "grillen", "."]
trg = ["two", "men", "barbecuing", "at", "a", "beach", "."]

print(f'src = {src}')
print(f'trg = {trg}')
```

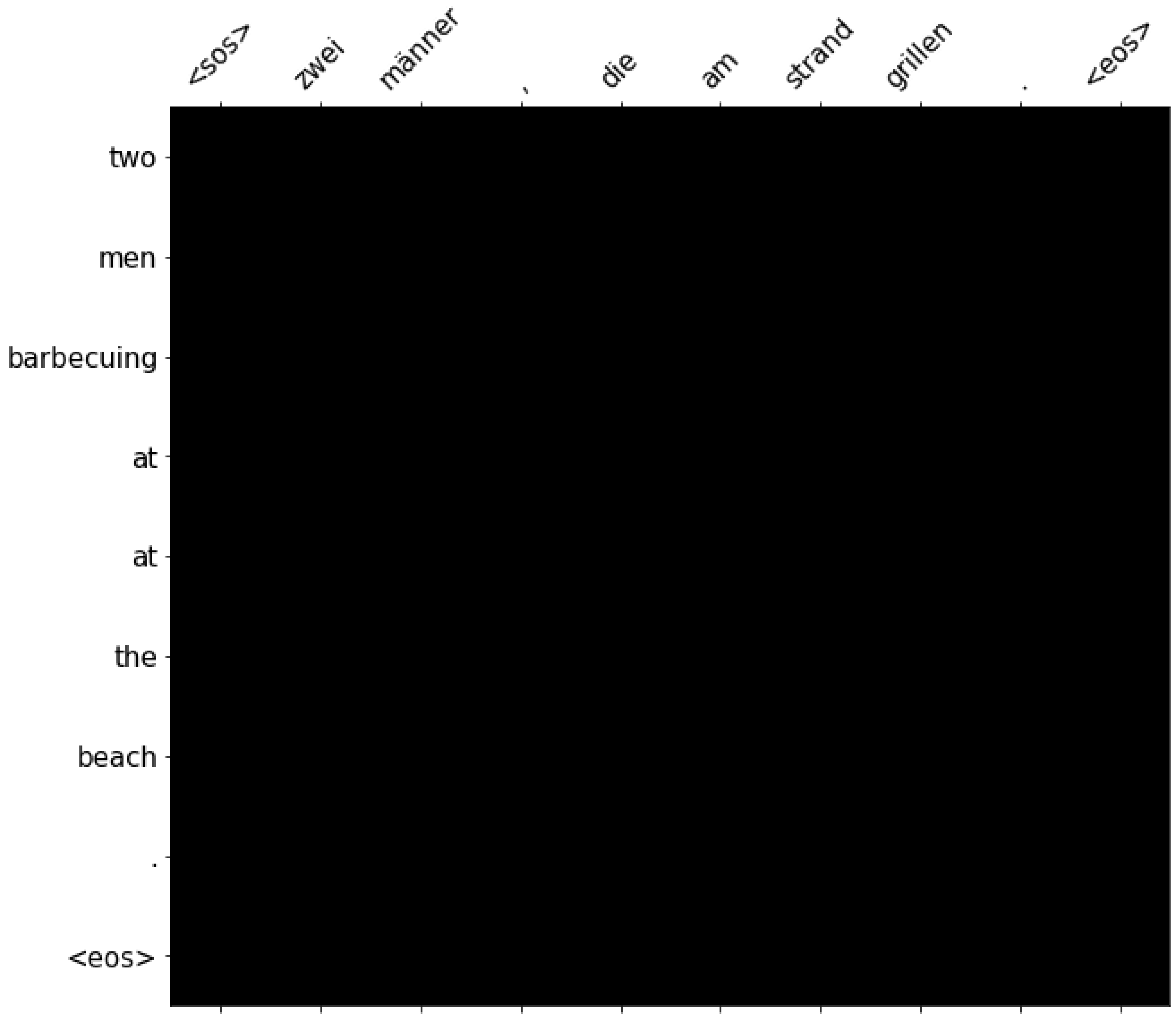
```
src = ['zwei', 'männer', ',', 'die', 'am', 'strand', 'grillen', '.']
trg = ['two', 'men', 'barbecuing', 'at', 'a', 'beach', '.']
```

Again, it produces a slightly different translation than target, a more literal version of the source sentence. It swaps *mountain climbing* for *climbing a mountain*.

In [43]:
```python
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')

display_attention(src, translation, attention)
```

```
predicted trg = ['two', 'men', 'barbecuing', 'at', 'at', 'the', 'beach', '.', '<eos>']
C:\Users\yoush\AppData\Local\Temp/ipykernel_19488/4268658779.py:15: UserWarning: FixedForm
atter should only be used together with FixedLocator
```

```
  ax.set_xticklabels(x_ticks, rotation=45)
C:\Users\yoush\AppData\Local\Temp/ipykernel_19488/4268658779.py:16: UserWarning: FixedForm
atter should only be used together with FixedLocator
  ax.set_yticklabels(y_ticks)
```



BLEU

Previously we have only cared about the loss/perplexity of the model. However there metrics that are specifically designed for measuring the quality of a translation - the most popular is *BLEU*. Without going into too much detail, BLEU looks at the overlap in the predicted and actual target sequences in terms of their n-grams. It will give us a number between 0 and 1 for each sequence, where 1 means there is perfect overlap, i.e. a perfect translation, although is usually shown between 0 and 100. BLEU was designed for multiple candidate translations per source sequence, however in this dataset we only have one candidate per source.

We define a `calculate_bleu` function which calculates the BLEU score over a provided TorchText dataset. This function creates a corpus of the actual and predicted translation for each source sentence and then calculates the BLEU score.

In [600…

```python
from torchtext.data.metrics import bleu_score

def calculate_bleu(data, src_field, trg_field, model, device, max_len = 50):

    trgs = []
    pred_trgs = []
    i = 1
    for datum in data:
        #print("current sentence: ", i)
        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = translate_sentence(src, src_field, trg_field, model, device, max_ler

        #cut off <eos> token
        pred_trg = pred_trg[:-1]
        #i+= 1
        pred_trgs.append(pred_trg)
        trgs.append([trg])

    #print(type(pred_trgs))
    #print(type(trgs))
    return bleu_score(pred_trgs, trgs)
```

We get a BLEU of around 28. If we compare it to the paper that the attention model is attempting to replicate, they achieve a BLEU score of 26.75. This is similar to our score, however they are using a completely different dataset and their model size is much larger - 1000 hidden dimensions which takes 4 days to train! - so we cannot really compare against that either.

This number isn't really interpretable, we can't really say much about it. The most useful part of a BLEU score is that it can be used to compare different models on the same dataset, where the one with the **higher** BLEU score is "better".

In [610…

```
%%time
bleu_score2 = calculate_bleu(test_data, SRC, TRG, model, device)

print(f'BLEU score = {bleu_score2*100:.2f}')
```

```
BLEU score = 29.62
Wall time: 19.5 s
```

In the next tutorials we will be moving away from using recurrent neural networks and start looking at other ways to construct sequence-to-sequence models. Specifically, in the next tutorial we will be using convolutional neural networks.

In [ ]:

```

```