AI539 Natural Language Processing with Deep Learning – Homework 3

Sequence-to-Sequence Models with Attention

Overview and Objectives. In this assignment, we'll get some hands-on experience with training recurrent neural networks for a conditional sequence generation task – translating German to English. This will be our first sequence-to-sequence model and the first time we are decoding language from an autoregressive RNN. Further, we will integrate a simple attention mechanism in the model to better understand their impact.

How to Submit. This assignment includes both coding questions and short response questions. Submit a zip file of your code and a PDF of your responses to Canvas. Do not include model checkpoints or training data.

Advice. Start early and be careful about shapes. For this assignment, we recommend getting on to the university high-performance cluster if you don't have a strong GPU at home. See Canvas page for details.

1 Attentive Sequence-to-Sequence Models for Machine Translation

Multi30k Dataset. Published in 2016, the Multi30k dataset supports machine translation between English and German (and later French). The dataset consists of roughly 30,000 image descriptions captured in both German and English by human translators. The skeleton code provides a convenient function to get dataloaders for all three splits and the training vocabulary for both German and English. Note how the vocabularies from `train_data` are passed to the validation and test dataset constructors. The vocabularies themselves offer `text2idx` and `idx2text` functions for numeralization and denumeralization operations which include language-specific tokenizers.

```
def getMulti30kDataloadersAndVocabs(batch_size=128):
  multi_train = Multi30kDatasetEnDe(split="train")
  multi_val = Multi30kDatasetEnDe(split="validation",
                                  vocab_en=multi_train.vocab_en,
                                  vocab_de=multi_train.vocab_de)
  multi_test = Multi30kDatasetEnDe(split="test",
                                   vocab_en=multi_train.vocab_en,
                                   vocab_de=multi_train.vocab_de)

  collate = Multi30kDatasetEnDe.pad_collate
  train_loader = DataLoader(multi_train, batch_size=batch_size,
                            num_workers=8, shuffle=True,
                            collate_fn=collate, drop_last=True)
  val_loader = DataLoader(multi_val, batch_size=batch_size,
                          num_workers=8,  shuffle=False,
                          collate_fn=collate)
  test_loader = DataLoader(multi_test, batch_size=batch_size,
                           num_workers=8, shuffle=False,
                           collate_fn=collate)

  return train_loader, val_loader,
         test_loader, {"en":multi_train.vocab_en, "de":multi_train.vocab_de}
```

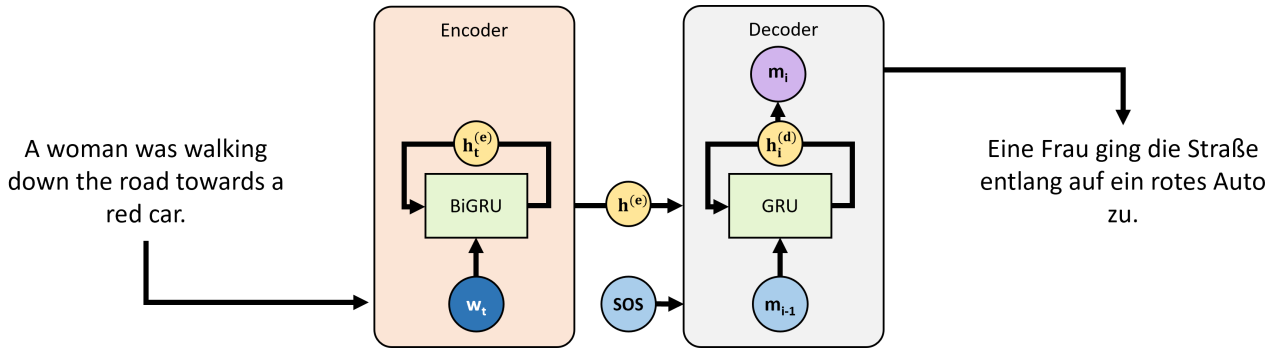
Similar to the previous assignment, the `pad_collate` function in `Multi30kDatasetEnDe` handles the variable length of input and output sequences – returning batches of padded, numeralized German (`xx_pad`) and English (`yy_pad`) sentences along with their lengths (`x_lens` and `y_lens`). Note that the dataset automatically prepends (appends) the English sentences with the `<SOS>` (`<EOS>`) so the decoder is trained to handle these tokens.

```python
@staticmethod
def pad_collate(batch):
    xx = [ele[0] for ele in batch]
    yy = [ele[1] for ele in batch]
    x_lens = torch.LongTensor([len(x)-1 for x in xx])
    y_lens = torch.LongTensor([len(y)-1 for y in yy])

    xx_pad = pad_sequence(xx, batch_first=True, padding_value=0)
    yy_pad = pad_sequence(yy, batch_first=True, padding_value=0)

    return xx_pad, yy_pad, x_lens, y_lens
```

Additional packages required. To support our dataset we'll need a couple more packages – specifically the `datasets` and `spacy` packages which you'll need to install via `pip`. Afterwards, you'll need to download the appropriate tokenizers from `spacy` by executing the following commands:

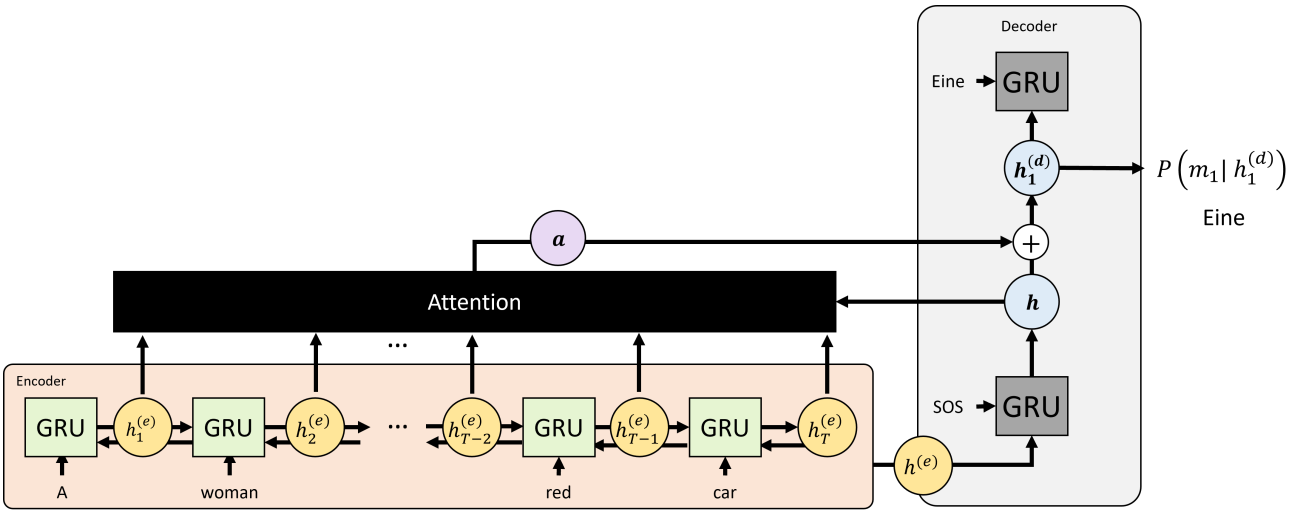
```
python -m spacy download en_core_web_sm
python -m spacy download de_core_news_sm
```

1.1 Sequence-to-Sequence Models



We are going to be considering a class of sequence-to-sequence models based on GRUs for this machine translation task – also known as encoder-decoder models. The simplest of these models (shown above) consists of an *encoder* bidirectional GRU that takes in a numeralized German sentence and a *decoder* unidirectional GRU that produces the corresponding English. During training, the decoder's hidden state is initialized by a representation from the encoder and then the decoder's outputs are supervised to assign high probability to the next word in the training English sentence via cross entropy loss.

As discussed in class, this basic structure relies on the encoder's output to capture everything relevant about the input and then the decoder's ability to retain that information over multiple timesteps. This can be difficult so adding an attention mechanisms that can "*look back*" at different parts of the input representation can potentially improve performance. We will consider different forms for this attention mechanism, but a general outline is shown below.



In this setting, the hidden state output h of our decoder GRU is used as a query to some attention process and the bidirectional hidden states of the encoder GRU are used as candidates. The resulting vector from attention is then added to h to make the attention-augmented state $h_1^{(d)}$ which is used to predict a distribution over the next word and is passed to the next time step of the decoder GRU.

Encoder. Let's formalize this a bit mathematically to be more concrete about what is happening. Consider a training example consisting of a source language sentence $w_1, ..., w_T$ and a target language sentence $m_1, ..., m_L$. Let $\mathbf{w}_t$ and $\mathbf{m}_i$ be one-hot word vectors such that

$$\mathbf{z_t} = \text{Dropout}(\mathbf{W}_e \mathbf{w}_t) \qquad \text{(Word Embedding)} \tag{1}$$

represents a word embedding with dropout applied. In PyTorch we will use word IDs and `nn.Embedding` layers rather than performing this sparse matrix-vector product to reduce computation. Given the word embeddings $z_1, ..., z_T$ for the source sentence, a bidirectional GRU computes hidden representations.

$$\overrightarrow{\mathbf{h}}_t^{(e)} = \overrightarrow{\text{GRU}}\left(\mathbf{z}_t, \overrightarrow{\mathbf{h}}_{t-1}^{(e)}\right) \qquad \text{(Forward GRU)} \tag{2}$$

$$\overleftarrow{\mathbf{h}}_t^{(e)} = \overleftarrow{\text{GRU}}\left(\mathbf{z}_t, \overleftarrow{\mathbf{h}}_{t+1}^{(e)}\right) \qquad \text{(Backward GRU)} \tag{3}$$

$$\mathbf{h}_t^{(e)} = \left[\overrightarrow{\mathbf{h}}_t^{(e)}, \overleftarrow{\mathbf{h}}_t^{(e)}\right] \forall t \qquad \text{(Word Representations)} \tag{4}$$

where $[\cdot, \cdot]$ denotes concatenation. While we write the forward and backward networks separately, PyTorch implements this as a single API call. To produce a sentence-level representation to initialize the decoder, we can take the final hidden states of the forward and backward GRUs:

$$\mathbf{h^{(e)}} = \left[\overrightarrow{\mathbf{h}}_T, \overleftarrow{\mathbf{h}}_0\right] \qquad \text{(Sentence Representation)} \tag{5}$$

▶ Q1 Implement Encoder [5pt]. In `models/Seq2SeqTranslator.py`, complete the implementation of the `BidirectionalEncoder` class by completing the functions shown below to implement the equations discussed in the previous section.

```
class BidirectionalEncoder(nn.Module):
    def __init__(self, src_vocab_len, emb_dim, enc_hid_dim, dropout=0.5):
        super().__init__()

        #TODO

    def forward(self, src, src_lens):

        #TODO

        return word_representations, sentence_rep
```

For a given source sequence of word IDs `src`, the model should embed each word ID to a `emb_dim` word embedding, apply Dropout with `dropout` probability to these embedding, then pass them through a 1-layer bidirectional GRU with hidden dimension of size `enc_hid_dim`. It should return the bidirectional hidden states for each word as a $B \times T \times 2*$`enc_hid_dim` tensor `word_representations`. The `sentence_rep` should be a $B \times 2*$`enc_hid_dim` tensor produced by concatenating the forward GRU hidden state for the last word in each sentence with the zeroth backward GRU hidden state for each sentence. Note, you'll need to use indexing to pick out the final forward states based on `src_lens`. Further, separating the forward/backward representations will require studying the GRU documentation in PyTorch.

To map the sentence representation to match the decoder hidden state size, our sequence-to-sequence model can include a linear layer and initialize the decoder hidden state with

$$\mathbf{h}_0^{(d)} = \text{GELU}(\mathbf{W}\mathbf{h}^{(e)} + \mathbf{b}) \qquad \text{(Initialize Decoder)} \tag{6}$$

For a standard seq-to-seq model without any attention mechanism, this would be all the information the decoder would get about the input sentence!

Decoder. Our decoder is a unidirectional GRU that performs an attention operation over the encoder word representations at each time step. For the ith word in the output sequence m_i, the decoder performs the following

operations:

$$\mathbf{b_i} = \text{Dropout}(\mathbf{W}_d\mathbf{m}_i) \quad \text{(Word Embedding)} \tag{7}$$

$$\mathbf{h} = \text{GRU}\left(\mathbf{b}_i, \mathbf{h}_{i-1}^{(d)}\right) \quad \text{(Forward GRU)} \tag{8}$$

$$\mathbf{a}_i = \text{Attn}\left(\mathbf{h}, \mathbf{h}_1^{(e)}, ..., \mathbf{h}_T^{(e)}\right) \quad \text{(Attention)} \tag{9}$$

$$\mathbf{h}_i^{(d)} = \mathbf{h} + \mathbf{a}_i \tag{10}$$

$$y = \mathbf{W}_d\mathbf{h}_i^{(d)} + \mathbf{b}_d \quad \text{(Output Scores)} \tag{11}$$

where the probability of the next word can be computed from the output scores via a softmax as $P(m_{i+1} \mid m_{\leq i}, w_1, ..., w_T) = \text{softmax}(y)$. We will leave the specifics of the `Attn` function to the next section and for now assume it returns some vector $\mathbf{a}_i$ the same dimension as $\mathbf{h}$.

▶ Q2 Implement Decoder [5pt]. In `models/Seq2SeqTranslator.py`, complete the implementation of the `Decoder` class by completing the functions shown below to implement the equations discussed in the previous section. Note that we are only considering the `forward` function for a single time step and will rely on the sequence-to-sequence model later to iterate through the target sentence.

```
class Decoder(nn.Module):
    def __init__(self, trg_vocab_len, emb_dim, dec_hid_dim, attention, dropout=0.5):
        super().__init__()

        self.attention = attention

        # TODO

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

        # TODO

        return hidden, out, alphas
```

For your implementation, assume that the provided `self.attention` supports the following operation:

```
attended_feature, alphas = self.attention(hidden, encoder_outputs)
```

where the `attended_feature` is the $B \times$ `dec_hid_dim` result of attention and `alpha` is the $B \times T$ tensor of attention values α over the input sequence. We will implement this in the next task.

The decoder itself will consist of word embeddings for the target language of dimension `emb_dim`, a unidirectional GRU with `dec_hid_dim` dimensional hidden states, and a classifier to predict word probabilities over the target vocabulary from the hidden state (`dec_hid_dim` dimensional linear layer, GELU, and a `trg_vocab_len` dimensional linear layer).

In the `forward` function, we will presume we are moving the decoder forward by only one time step. As such, the `input_word` is a B dimensional tensor of word IDs, `hidden` is a $B \times$ `dec_hid_dim` tensor containing the previous hidden states, and `encoder_outputs` is the $B \times T \times 2*$`enc_hid_dim` `word_representations` tensor returned from the `BidirectionalEncoder`. With these inputs, the `forward` function should embed the input words, apply dropout, and then step the GRU forward to produce an updated hidden state ($\mathbf{h}$). Then the attention function should be called to produce the attended feature and alphas. The updated hidden state and attended feature should be added to produce the new hidden state and the classifier should be applied to produce a $B \times$ `trg_vocab_len` tensor of word scores. The new hidden state, this output tensor, and the attention alphas should be returned.

Attention. Recall from the lecture the definition of a single-query scaled dot-product attention mechanism. Given a query $\mathbf{q} \in \mathbb{R}^{1 \times d_{kq}}$, a set of candidates represented by keys $\mathbf{k}_1, ..., \mathbf{k}_T \in \mathbb{R}^{1 \times d_{kq}}$ and values $\mathbf{v}_1, ..., \mathbf{v}_T \in \mathbb{R}^{1 \times d_v}$, we

compute the scaled dot-product attention (with the softmax explicitly written out) as:

$$\alpha_i = \frac{\exp\left(\mathbf{q}\mathbf{k}_i^T / \sqrt{d_{kq}}\right)}{\sum_{j=1}^{T} \exp\left(\mathbf{q}\mathbf{k}_j^T / \sqrt{d_{kq}}\right)} \quad \forall i \tag{12}$$

$$\mathbf{a} = \sum_{j=1}^{T} \alpha_j \mathbf{v}_j \tag{13}$$

where the α_i are collectively referred to as the attention distribution and $\mathbf{a}$ the attended feature.

▶ Q3 Implement Attention Mechanism [10pt]. Complete the `DotProductAttention` class by completing the following functions in `models/Seq2SeqTranslator.py` to implement a batched version of the attention mechanism described above where the queries are generated from `hidden` state of our decoder and the keys / values from the `encoder_outputs`.

```
class DotProductAttention(nn.Module):

    def __init__(self, q_input_dim, cand_input_dim, v_dim, kq_dim=64):
        super().__init__()

        #TODO

    def forward(self, hidden, encoder_outputs):

        #TODO

        return attended_val, alphas
```

To transform the inputs to queries, keys, and values we will need three linear transformations: one to convert the `q_input_dim` dimensional `hidden` input to `kq_dim`, one to convert the `cand_input_dim` dimensional `encoder_outputs` input to `kq_dim`, and one to convert the `cand_input_dim` dimensional `encoder_outputs` input to `v_dim`.

In the `forward` function, the following operations should be performed

- The `hidden` input should be transformed to a $B \times$ `kq_dim` `query` tensor via a linear layer.
- The `encoder_outputs` input should be converted a to $B \times T \times$ `kq_dim` `key` tensor via a linear layer.
- The `encoder_outputs` input should be converted a to $B \times T \times$ `v_dim` `value` tensor via a linear layer.
- Use torch.bmm and appropriate `permute` operations to compute a $B \times 1 \times T$ output resulting from computing scaled dot products between each of the B query and key matrix pairs. Be sure to scale the dot products by the square root of `kq_dim`.
- Apply squeeze and softmax operations to produce an $B \times T$ `alpha` tensor from the scaled dot product tensor wherein each row sums to 1.
- Use torch.bmm and appropriate `permute` / `unsqueeze` operations to compute a $B \times$ `v_dim` attended feature tensor.

Return the attended feature and alpha tensors.

In addition to the attention mechanism you've implemented, the code provides two more – a `Dummy` attention that always returns a zero vector and a `MeanPool` attention that assigns equal attention weight to all inputs. We will examine the impact of each later on in the assignment. The default hyperparameters use the `DotProductAttention` you implemented above.

Our full model. To put these modules together, we define a `Seq2Seq` model, the `init` of which can be seen below.

```
class Seq2Seq(nn.Module):
    def __init__(self, src_vocab_size, trg_vocab_size, embed_dim, enc_hidden_dim,
                 dec_hidden_dim, kq_dim, attention, dropout=0.5):
        super().__init__()

        self.trg_vocab_size = trg_vocab_size
        self.encoder = BidirectionalEncoder(src_vocab_size, embed_dim,
                                            enc_hidden_dim, dropout)
        self.enc2dec = nn.Sequential(nn.Linear(enc_hidden_dim*2,dec_hidden_dim), nn.GELU())

        if attention == "none":
            attn_model = Dummy(dec_hidden_dim)
        elif attention == "mean":
            attn_model = MeanPool(2*enc_hidden_dim, dec_hidden_dim)
        elif attention == "dotproduct":
            attn_model = DotProductAttention(dec_hidden_dim, 2*enc_hidden_dim,
                                             dec_hidden_dim, kq_dim)

        self.decoder = Decoder(trg_vocab_size, embed_dim, dec_hidden_dim,
                               attn_model, dropout)
```

It already constructs the encoder, a linear layer to convert the `sentence_rep` output from the encoder to the size of the decoder hidden state, and a decoder with the requested attention mechanism. We need to finish writing its forward pass and provide a utility function to decode outputs autoregressively at test time.

▶ Q4 Implement Seq2Seq Model Forward [5pt]. Complete the Seq2Seq model by implementing the following function in `models/Seq2SeqTranslator.py`. This function will be used during training to compute predicted scores for each next word in a training example.

```
class Seq2Seq(nn.Module):

    . . .

    def forward(self, src, trg, src_lens):
        #tensor to store decoder output scores
        outputs = torch.zeros(trg.shape[0], trg.shape[1],
                              self.trg_vocab_size).to(src.device)

        #TODO

        return outputs
```

In the `forward` function, the source sentence `src` should be processed by the `encoder` to yield the `word_representations` and `sentence_rep` tensors. The `enc2dec` layer should be used to initialize the hidden states of the `decoder` with transformed `sentence_rep`. Then, for each time step t in the target sentence `trg` of length L,

- The decoder model should be applied to step forward one step – taking the ground truth `trg` words at time t as input and updating the hidden state and producing an $B \times$ `trg_vocab_size` output tensor of next-word scores for each vocabulary word.
- These predicted scores should be stored in the $B \times L \times$ `trg_vocab_size` `outputs` tensor at the appropriate location based on time step.

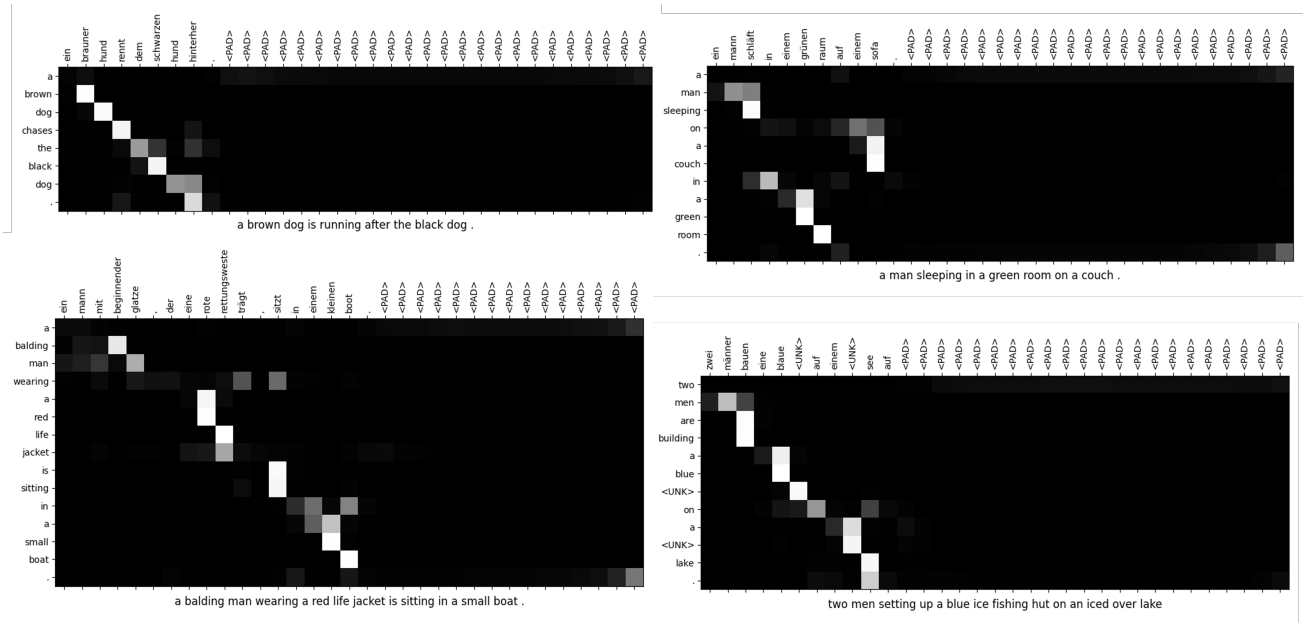
This process will require the use of a `for` loop over the time dimension. After processing the entire target sequence, the output tensor should be returned. The cross entropy loss in `train.py` will then optimize model weights to increase the probability of the ground truth words.

▶ Q5 Implement Seq2Seq Translation [5pt]. Complete the Seq2Seq model by implementing the following function to enable autoregressive generation from our model. When translating a new sentence, we only have access to the source language and need to alter our logic from the `forward` function to enable autoregressive generation.

```
class Seq2Seq(nn.Module):

    . . .

    def translate(self, src, src_lens, sos_id=1, max_len=50):
        #tensors to store decoder output word IDs and attention matrices
        outputs = torch.zeros(src.shape[0], max_len).to(src.device)
        attns = torch.zeros(src.shape[0], max_len, src.shape[1]).to(src.device)

        # Tensor of initial <SOS> inputs
        input_words = torch.ones(src.shape[0], dtype=torch.long).to(src.device)
        input_words = input_words*sos_id

        #TODO

        return outputs, attns
```

This `translate` function differs from `forward` in two major ways: (1) rather than a ground truth `trg` word at each time step, the input words for the next time step should be those with the highest predicted scores from the model (i.e., argmax decoding), and (2) we would like to also return the attention matrices produced by the decoder for visualization. The skeleton code already provides `output` and `attns` tensors to store the generated word IDs and attention matrices. Further, we start the `input_words` off as a tensor of IDs corresponding to the <SOS> token. Autoregressive generation should continue for `max_len` time steps.

If everything is implemented correctly, you should be able to run `train.py` to build your model and execute training – this should take roughly 30 minutes on a GPU. Logging to `wandb` has already been setup and correct models should be able to achieve per-word validation accuracies near 60-62%. The BLEU score for the validation set is also computed and logged. Further, the training scripts also logs attention distribution visualizations for 10 examples from the validation set like those shown below.



Each shows the argmax predicted translation for a validation input sentence along with a visualization of the attention distributions. In each, the input German sentence is shown word-by-word along the top axis and the model-generated translation is shown word-by-word along the left axis. Each row shows the attention distribution (`alphas`) at that time step where brighter values are nearer to 1. The ground truth translation is shown below the figure. The top-right example is a particularly interesting case where the inversion of "on a couch" and "in a green room" between the two languages can be observed in the attention patterns. It is worth reflecting on the fact that the model was never given any explicit guidance about how these patterns should look, rather they emerge as part of minimizing prediction error for the English sentences.

▶ Q6 Train model and examine attention patterns [5pt]. Train your model with default hyperparameters and provide the training and validation plots in your report. Also provide and discuss two attention visualizations.

▶ Q6 Report In this section, we present the training and validation results obtained using the default hyperparameters 1. We also discuss two examples of attention visualizations to understand how the model aligns the input and output sequences during translation.

During training, we monitored the training and validation loss and accuracy. As can be seen in Figure 2 and 3, The model converged after approximately 8k epochs, achieving a validation accuracy of 62.508%.

We selected two examples from the validation set to analyze the learned attention patterns 4. In the first example (left), the input German sentence ("ein brauner hund rennt dem schwarzen hund hinterher") is translated into English as "a brown dog running after black dog." This example illustrates that while model largely learns direct alignments, it can also differ between source and target languages.

In the second example (right), the input German sentence ("ein junge mit kopfhorern sitzt auf den schultern einer frau") is translated into English as "a boy with earphones sitting on the shoulders of a woman." This strong diagonal suggests a near one-to-one translation structure where the sentence between German and English is relatively parallel.

```
config = {
    "bs":128,   # batch size
    "lr":0.001,# learning rate
    "l2reg":0.0000001,# weight decay
    "max_epoch":50,
    "embed_dim":128,
    "enc_dim":256,
    "dec_dim":256,
    "kq_dim":64,
    "attn": "dotproduct", #Options are none, mean, dotproduct
    "dropout":0.5
}
```

Figure 1: Default Hyperparameters

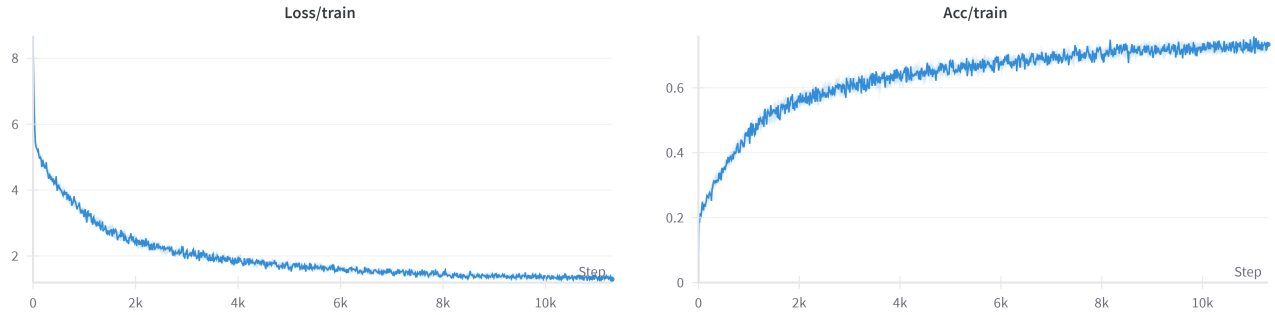


Figure 2: Training Loss (left) and Accuracy (right)

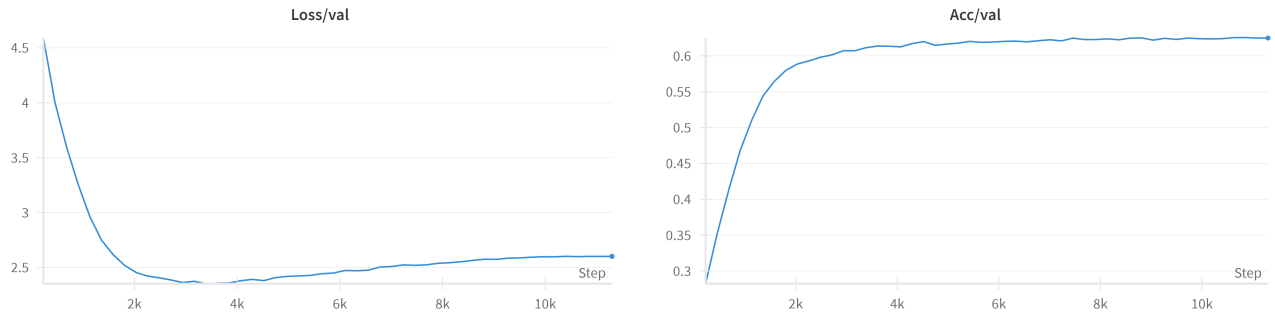


Figure 3: Validation Loss (left) and Accuracy (right)

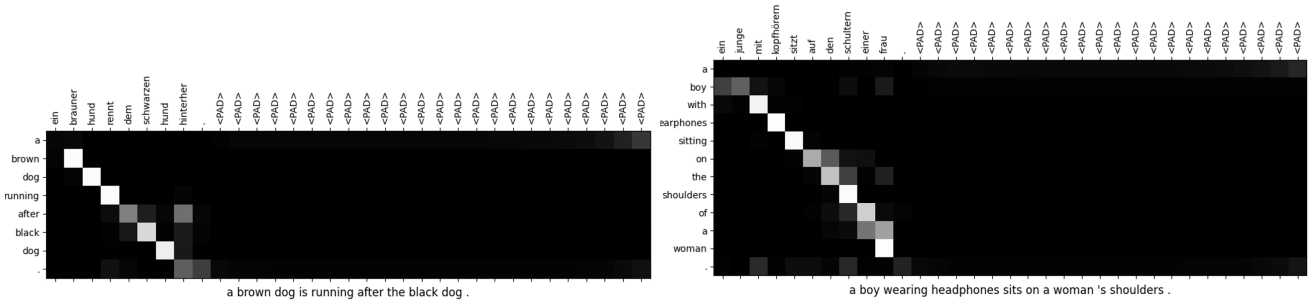


Figure 4: Two attention visualizations: dog (left) and boy (right)

2 Experimentation

▶ Q7 Experiment with attention mechanisms [3pt]. Rerun your training experiments while setting the ‘‘attn’’ value in the configuration to “none” and “mean” to use no attention mechanism and one which averages the input encoders `word_representations` in place of the dot product attention. Provide your plots from `wandb` comparing these to our default attention-based model. Describe what you observe.

▶ Q7 Report We conducted experiments using three different attention mechanisms to evaluate their impact on translation quality: Dot product, Mean pooling ("mean"), and Dummy ("none") attention.

As can be seen in Figure 6, Dot product attention (purple) achieved the best performance by allowing the decoder to dynamically focus on relevant parts of the input at each step. Mean pooling attention (green) achieved less performance compared to Dot product attention. It lacked the precision needed for complex sentence structures. Dummy attention (blue) resulted in severely degraded performance because the decoder had no information about the input sentence.

Figure 7 shows the three different attention models of BLEU scores and two visualizations ("mean pooling" and "none"). Dot product also achieved the best score, approximately 0.35. On the other hand, Mean pooling and Dummy attention models are lower than 0.25. Without a proper attention model, mean pooling and dummy attention models resulted in nothing on each visualization.

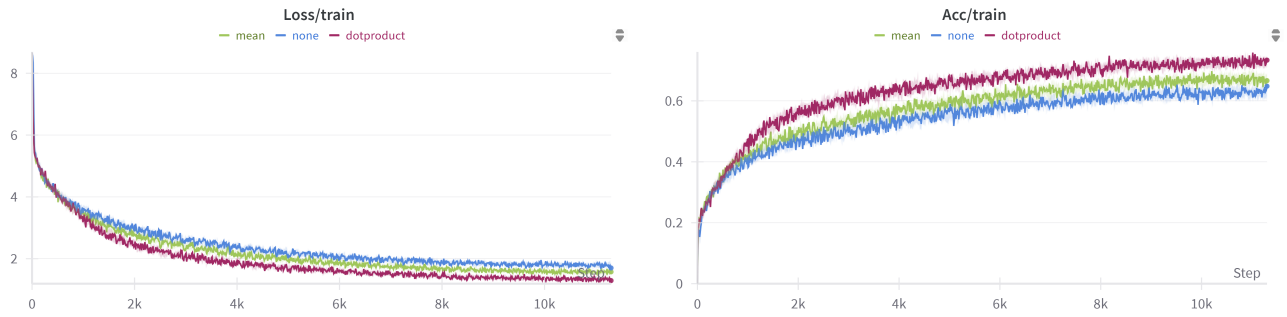


Figure 5: Train Loss (left) and Accuracy (right)

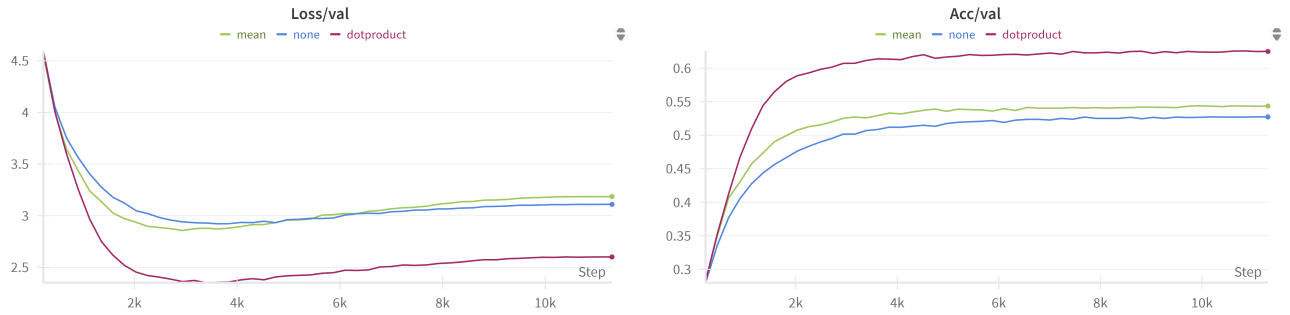


Figure 6: Validation Loss (left) and Accuracy (right)

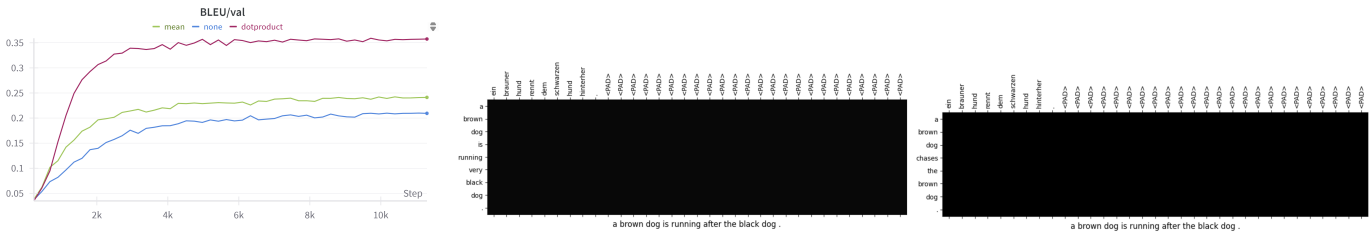


Figure 7: Comparison of three BLEU scores (left) and visualizations (mid, right)

▶ Q8 Experiment with dropout [2pt]. With the default configuration, set the "dropout" parameter to 0 such that the Dropout layers have no effect during training or inference. Provide your plots from wandb comparing these to our default dropout probability of 0.5. Describe what you observe.

▶ Q8 Report To evaluate the impact of dropout on model performance, we trained two models: 0.5 dropout and 0 dropout. All other hyperparameters were kept default settings.

In Figure 9 When dropout is 0.5, the validation accuracy is nearly 62%, whereas the validation accuracy is about 58% when dropout is 0. This means that the model trained without dropout achieved lower validation accuracy. Figure 8 and 9 shows that training loss decreased quickly for both models, but the no-dropout model had a large gap between training and validation loss (a signal of overfitting). In particular, validation loss dramatically increased after 2k iterations until reaching 5.

As shown in Figure 10, there are two different dropout style of BLEU scores. At 2k iterations, 0.5 dropout model (0.35) achieved better performance than no-dropout one (0.3). Interestingly, the visualizations of the sentences with no-dropout are well-aligned with ground-truth sentences, rather than 0.5 dropout.

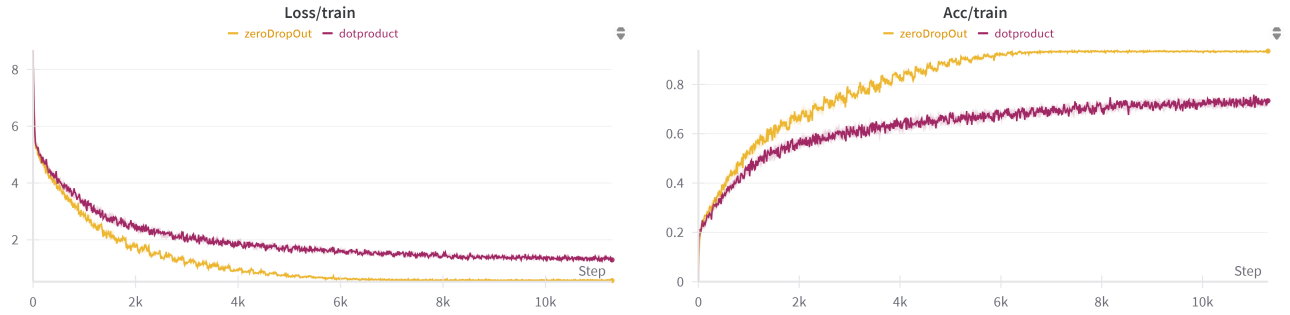


Figure 8: Train loss (left) and Accuracy (right)

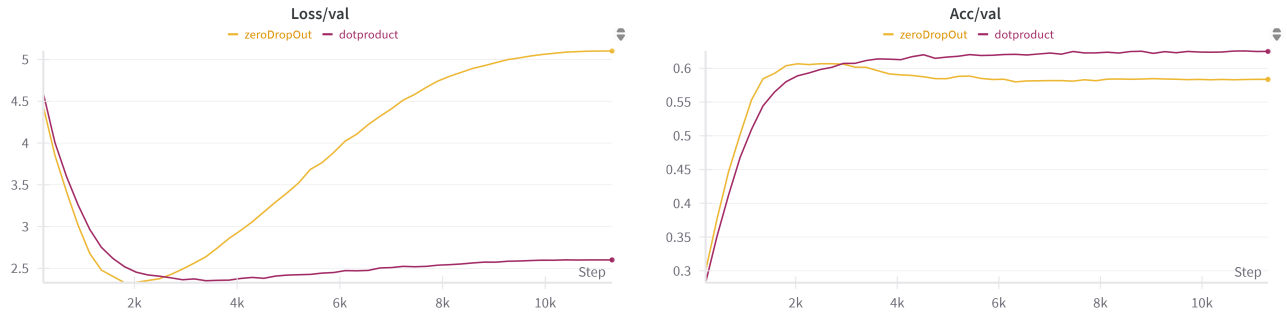


Figure 9: Validation loss (left) and Accuracy (right)

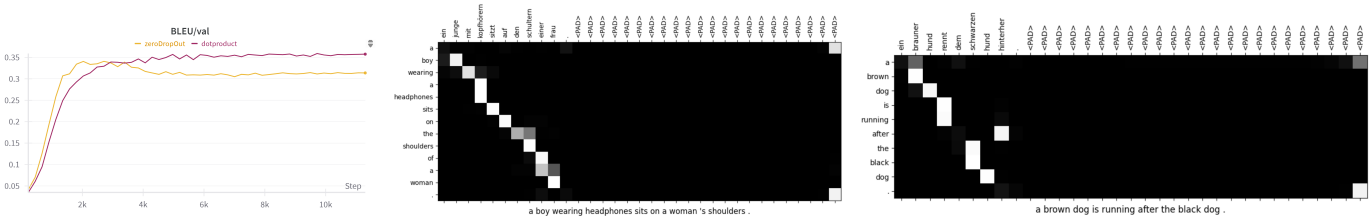


Figure 10: BLEU scores (left) and visualization (right)

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
   ⇒ I spent 5 days on this assignment
2. Would you rate it as easy, moderate, or difficult?
   ⇒ between moderate and difficult
3. Did you work on it mostly alone or did you discuss the problems with others?
   ⇒ I worked on it mostly alone.
4. How deeply do you feel you understand the material it covers (0%–100%)?
   ⇒ 70 - 80%
5. Any other comments?
   ⇒ Thanks for your hard work and time.