

Attention Mechanisms in Sequence-to-Sequence Models

Overview and Objectives. In this homework, we'll build some intuitions for scaled dot-product attention and implement a simple attention mechanism for an RNN-based sequence-to-sequence model in a small-scale machine translation task. If you finish early, *work on your projects!*

How to Do This Assignment. The assignment has a few math questions and then walks you through completing the provided skeleton code and analyzing some of the results. Anything requiring you to do something is marked as a "Task" and has associated points listed with it. You are expected to turn in both your code and a write-up answering any task that requested written responses. Submit a zip file containing your completed skeleton code and a PDF of your write-up to Canvas.

Advice. Start early. Students will need to become familiar with `pytorch` for this and future assignments. Extra time may be needed to get used to working remotely on the GPU cluster here. You can also use GPU-enabled runtimes in Colab colab.research.google.com.

1 Scaled Dot-Product Attention [8 pts]

To warm up and start getting more familiar with scaled dot-product attention mechanisms, we'll do some exploratory math first.¹ Recall from the lecture the definition of a single-query scaled dot-product attention mechanism. Given a query $\mathbf{q} \in \mathbb{R}^{1 \times d}$, a set of candidates represented by keys $\mathbf{k}_1, \dots, \mathbf{k}_m \in \mathbb{R}^{1 \times d}$ and values $\mathbf{v}_1, \dots, \mathbf{v}_m \in \mathbb{R}^{1 \times d_v}$, we compute the scaled dot-product attention as:

$$\alpha_i = \frac{\exp(\mathbf{q}\mathbf{k}_i^T / \sqrt{d})}{\sum_{j=1}^m \exp(\mathbf{q}\mathbf{k}_j^T / \sqrt{d})} \quad (1)$$

$$\mathbf{a} = \sum_{j=1}^m \alpha_j \mathbf{v}_j \quad (2)$$

where the α_i are referred to as attention values (or collectively as an attention distribution). The goal of this section is to get a feeling for what is easy for attention to compute and why we might want something like multi-headed attention to make some computations easier.

► **TASK 1.1 Copying [2pts]** Describe (in one or two sentences) what properties of the keys and queries would result in the output \mathbf{a} being equal to one of the input values \mathbf{v}_j . Specifically, what must be true about the query \mathbf{q} and the keys $\mathbf{k}_1, \dots, \mathbf{k}_m$ such that $\mathbf{a} \approx \mathbf{v}_j$? (We assume all values are unique – $\mathbf{v}_i \neq \mathbf{v}_j, \forall i \neq j$.)

► **TASK 1.2 Average of Two [2pts]** Consider a set of key vectors $\{\mathbf{k}_1, \dots, \mathbf{k}_m\}$ where all keys are orthogonal unit vectors – that is to say $\mathbf{k}_i \mathbf{k}_j^T = 0, \forall i \neq j$ and $\|\mathbf{k}_i\| = 1, \forall i$. Let $\mathbf{v}_a, \mathbf{v}_b \in \{\mathbf{v}_1, \dots, \mathbf{v}_m\}$ be two value vectors. Give an expression for a query vector \mathbf{q} such that the output \mathbf{a} is approximately equal to the average of \mathbf{v}_a and \mathbf{v}_b , that is to say $\mathbf{a} \approx \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$. You can reference the key vectors corresponding to \mathbf{v}_a and \mathbf{v}_b as \mathbf{k}_a and \mathbf{k}_b respectively. Note that due to the softmax in Eq. 1, it won't ever actually reach this value, but you can make it arbitrarily close by adding a scaling constant to your solution.

In the previous task, we saw it was *possible* for a single-headed attention to focus equally on two values. This can easily be extended to any subset of values. In the next question we'll see why it may not be a *practical* solution.

¹These questions are adapted from CS224n at Stanford because I just liked the originals too much not to use them.

► **TASK 1.3 Noisy Average [2pts]** Now consider a set of key vectors $\{\mathbf{k}_1, \dots, \mathbf{k}_m\}$ where keys are randomly scaled such that $\mathbf{k}_i = \mu_i * \lambda_i$ where $\lambda_i \sim \mathcal{N}(1, \beta)$ is a randomly sampled scalar multiplier. Assume the unscaled vectors μ_1, \dots, μ_m are orthogonal unit vectors. If you use the same strategy to construct the query q as you did in Task 1.2, what would be the outcome here? Specifically, derive $\mathbf{q}\mathbf{k}_a^T$ and $\mathbf{q}\mathbf{k}_b^T$ in terms of μ 's and λ 's. Qualitatively describe what how the output a would vary over multiple resamplings of $\lambda_1, \dots, \lambda_m$.

As we just saw in 1.3, for certain types of noise that either scale (shown here) or change the orientation of the keys, single-head attention may have difficulty combining multiple values consistently. Multi-head attention can help.

► **TASK 1.4 Noisy Average with Multi-head Attention [2pts]** Let's now consider a simple version of multi-head attention that averages the attended features resulting from two different queries. Here, two queries are defined (\mathbf{q}_1 and \mathbf{q}_2) leading to two different attended features (\mathbf{a}_1 and \mathbf{a}_2). The output of this computation will be $\mathbf{a} = \frac{1}{2}(\mathbf{a}_1 + \mathbf{a}_2)$. Assume we have keys like those in Task 1.3, design queries \mathbf{q}_1 and \mathbf{q}_2 such that $\mathbf{a} \approx \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$.

2 Attention in German-to-English Machine Translation [12 pts]

In this part of the homework, we are going to get some experience implementing attention mechanisms on top of familiar components. In HW2 we used bidirectional encoders for Part-of-Speech tagging and in HW3 we decoded unconditional language models. Here we'll combine these into a sequence-to-sequence model that translates German sentences to English. The skeleton code already provides the data loading, tokenization, training / evaluation infrastructure, and encoder/decoder model structure. The skeleton code has been provided as a Colab Notebook.

To keep training time short (~5-10 minutes), we are using a small-scale translation datasets called Multi30k [1] that contains over 31,014 bitext sentences describing common visual scenes in both German and English (split across train, val, and test). It is intended to support multimodal translation (i.e. utilizing an image of the described scene to make translation easier) but we will just use it as a text problem for simplicity.

Students can look through the provided code for the implementation details; however, the computation our model is performing is summarized in the following equations. Consider a training example consisting of a source language sentence w_1, \dots, w_T and a target language sentence m_1, \dots, m_L . Let \mathbf{w}_t and \mathbf{m}_t be one-hot word encodings.

Encoder. Our encoder is a simple bidirectional GRU model. While we write the forward and backward networks separately, PyTorch implements this as a single API.

$$\mathbf{z}_t = \mathbf{W}_e \mathbf{w}_t \quad (\text{Word Embedding}) \quad (3)$$

$$\vec{\mathbf{h}}_t^{(e)} = \text{GRU} \left(\vec{\mathbf{z}}_t, \vec{\mathbf{h}}_{t-1}^{(e)} \right) \quad (\text{Forward GRU}) \quad (4)$$

$$\overleftarrow{\mathbf{h}}_t^{(e)} = \text{GRU} \left(\overleftarrow{\mathbf{z}}_t, \overleftarrow{\mathbf{h}}_{t+1}^{(e)} \right) \quad (\text{Backward GRU}) \quad (5)$$

$$\mathbf{h}_t^{(e)} = \begin{bmatrix} \vec{\mathbf{h}}_t^{(e)} & \overleftarrow{\mathbf{h}}_t^{(e)} \end{bmatrix} \quad \forall t \quad (\text{Word Representations}) \quad (6)$$

$$\mathbf{h}_{\text{sent.}}^{(e)} = \text{ReLU} \left(\mathbf{W}_e \begin{bmatrix} \vec{\mathbf{h}}_T & \overleftarrow{\mathbf{h}}_0 \end{bmatrix} + \mathbf{b}_e \right) \quad (\text{Sentence Representation}) \quad (7)$$

Decoder. Our decoder is a unidirectional GRU that performs an attention operation over the encoder word representations at each time step (Eq. 12). Notice that we initialize the decoder hidden state with the overall sentence encoding from the encoder.

$$\mathbf{h}_0^{(d)} = \mathbf{h}_{sent}^{(e)} \quad (\text{Initialize Decoder}) \quad (8)$$

$$\mathbf{b}_i = \mathbf{W}_d \mathbf{m}_i \quad (\text{Word Embedding}) \quad (9)$$

$$\mathbf{h}_i^{(d)} = \text{GRU}(\mathbf{b}_i, \mathbf{h}_{i-1}^{(d)}) \quad (\text{Forward GRU}) \quad (10)$$

$$\mathbf{a}_i = \text{Attn}(\mathbf{h}_i^{(d)}, \mathbf{h}_1^{(e)}, \dots, \mathbf{h}_T^{(e)}) \quad (\text{Attention}) \quad (11)$$

$$P(m_{i+1} | m_{\leq i}, w_1, \dots, w_T) = \text{softmax}(\mathbf{W}_d [\mathbf{a}_i, \mathbf{h}_i^{(d)}] + \mathbf{b}_d) \quad (\text{Prob. of Next Word}) \quad (12)$$

Our explorations in this assignment will be implementing and contrasting different choices for the $\text{Attn}(q, c_1, \dots, c_T)$ module above. All the other elements of the encoder-decoder architecture have already been implemented.

► **TASK 2.1 Scaled-Dot Product Attention [8pts]** Implement $\text{Attn}(\cdot)$ in equation (11) as single-query scaled dot-product attention as defined in equations (1) and (2). Here, the query will be the decoder hidden state and the keys and values will be derived from the encoder representations. Implement this attention mechanism by completing the `SingleQueryScaledDotProductAttention` class. The skeleton code is below:

```
1 class SingleQueryScaledDotProductAttention(nn.Module):
2     # kq_dim is the dimension of keys and queries. Linear layers should be used
3     # to project inputs to these dimensions.
4     def __init__(self, enc_hid_dim, dec_hid_dim, kq_dim=512):
5         super().__init__()
6         .
7         .
8         .
9         #hidden is h_t^{(d)} from Eq. (11) and has dim => [batch_size, dec_hid_dim]
10        #encoder_outputs is the word representations from Eq. (6)
11        # and has dim => [src_len, batch_size, enc_hid_dim * 2]
12        def forward(self, hidden, encoder_outputs):
13            .
14            .
15            .
16            .
17            .
18            assert attended_val.shape == (hidden.shape[0], encoder_outputs.shape[2])
19            assert alpha.shape == (hidden.shape[0], encoder_outputs.shape[0])
20            return attended_val, alpha
```

The forward function takes two inputs – `hidden` is the decoder hidden state $\mathbf{h}_j^{(d)}$ and `encoder_outputs` corresponds to encoder word representations $\mathbf{h}_t^{(e)}$, $\forall t$. These should be converted to keys, queries, and values:

$$\mathbf{q} = \mathbf{W}_q \mathbf{h}_j^{(d)} \quad (13)$$

$$\mathbf{k}_t = \mathbf{W}_k \mathbf{h}_t^{(e)} \quad (14)$$

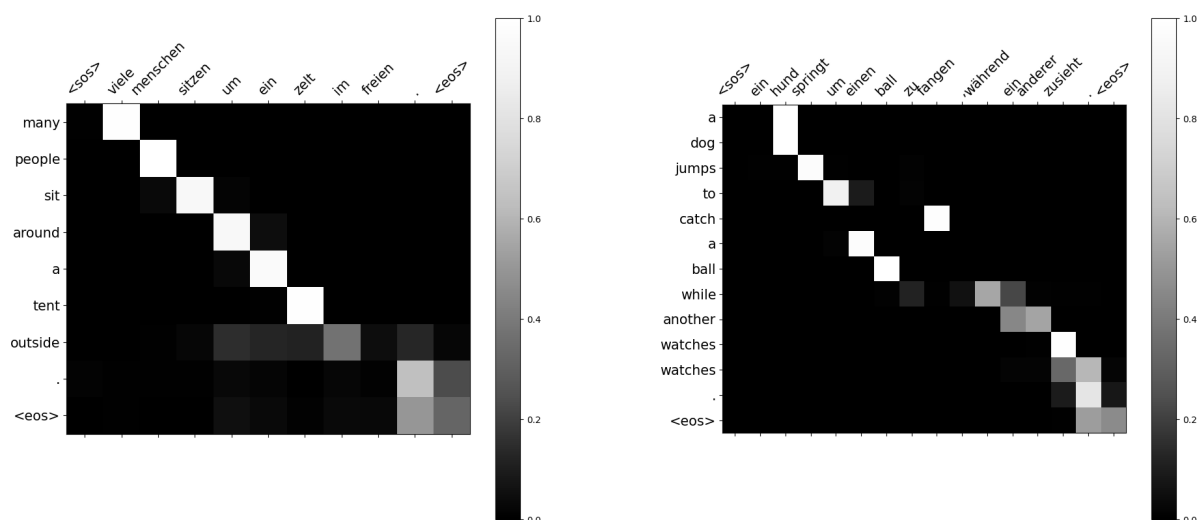
$$\mathbf{v}_t = \mathbf{h}_t^{(e)} \quad (15)$$

And the output – `attended_val` and `alpha` – correspond to the attended value vector (\mathbf{a}) and the vector of attention values (α) computed from as in equations (1) and (2). The expected dimensions are asserted above. Note that this is intended to be a batched operation and the equations presented are for a single instance. `torch.bmm` can be very useful here.

Train this model then record the perplexity and BLEU score on the test set – BLEU values of around 30-35 are expected for correct implementations. These are automatically generated in the script and printed after training.

After implementing the scaled dot-product attention mechanism, running the notebook will execute training the model with your attention mechanism. The code saves the checkpoint with the lowest training validation. Afterwards, it

will report BLEU on the test set as well as produce a number of examples (printed in console) with attention diagrams (saved in `examples` folder) like those shown below



where the brighter blocks indicate high attention values over source word encodings (columns) used when generating translated words (rows). Remember that these encodings also carry information about the rest of the sentence as they come from the bidirectional encoder.

► **TASK 2.2 Attention Diagrams [1pts]** Search through the attention diagrams produced by your model. Include a few examples in your report and characterize common patterns you observe. Note that German is (mostly) a Subject-Object-Verb language so you may find attention patterns that indicate inversion of word order when translating to Subject-Verb-Object English as in the 2nd example above.

The code also implements two baseline options for the attention mechanism. A *Dummy* attention that just outputs zero tensors – this effectively attends to *no* words. The *MeanPool* attention which just averages the source word encodings – this effectively attends *equally* to all words. Adjusting the `attn` variable can change to these options.

► **TASK 2.3 Comparison [3pts]** Train and evaluate models with the *Dummy* and *MeanPool* ‘attention’ mechanisms. Report mean and variance over three runs for these baselines and your implementation of scaled dot-product attention. Discuss the observed trends.

► **TASK 2.EC Beam Search and BLEU [3pts]** This is an extra credit question and is optional.

In the previous homework, we implemented many decoding algorithms; however, in this work we just use greedy top-1 in the `translate_sentence` function. Adapt your implementation of beam search from HW3 to work on this model by augmenting `translate_sentence` (which is used when computing BLEU). Report BLEU scores on the test set for the scaled dot-product attention model with $B=5, 10, 20$, and 50 .

References

- [1] D. Elliott, S. Frank, K. Sima'an, and L. Specia, “Multi30k: Multilingual english-german image descriptions,” in *Proceedings of the 5th Workshop on Vision and Language*, pp. 70–74, 2016.