Attention-Based Models

Muchang Bahng

Spring 2023

Contents

1 Attention Models

We have seen the power of encoder decoder models, which can be used with a combination of RNNs or CNNs. Since we can plug and play different architectures, we can process and output different types of data.[1] CNNs were quite strong and have really no problem with scaling ever since the ResNet architecture, but historically, RNNs had two main problems.

1. They compute sequentially, since the hidden states must be a function of previous hidden states. This results in a linear time complexity, which is not ideal.
2. By encoding the entire sequence in a hidden latent space of dimension h, we are essentially trying to compress a possibly very long sequence into a single vector of predetermined dimension. This causes a *bottleneck problem* and words that are further away may be "forgotten."

We will see that attention solves the second bottleneck problem, and self-attention solves the sequential problem.

In general, feed forward networks treat features as independent, convolutional networks focus on relative location and proximity, and RNNs have tend to read in one direction. This may not be the most flexible way to process data, and we have some other problems.

1. When processing images, we may want our CNN to focus on a specific part of the image. For example, when we see a cat in the corner, other parts of the image does not matter, and we can have our CNN focus on the specific portion of the image containing the cat.
2. When reading sentences, different words may be interdependent, even if they are not next to each other, and so we may want to focus on different portions of a sentence (e.g. words 1 3, plus 10 15 which describes an object).

This is where attention comes in to the rescue. Attention is, to some extent, motivated by how we pay visual attention to different regions of an image or correlate words in one sentence. Human attention allows us to focus on certain regions or portions of our data with "high resolution" while perceiving the surrounding data in "low resolution." In a nutshell, attention in deep learning can be broadly interpreted as a vector of importance weights. First, we will introduce attention in the general setting, then move onto its specific implementation in RNNs through the seq2seq attention model, and finally its application in CNNs through Vision transformers.

1.1 Seq2Seq with Attention

The idea of **attention** provides a solution to this bottleneck problem. Basically, we want to establish connections from the decoder to not just the last hidden state of the encoder, but to all of its nodes. Each encoder node represents some information about each word, and by taking some weighted sum of these nodes, we can choose which one to put this attention on.

Definition 1.1 (Score Function)

Before we begin, let's define a metric, called a **score function**, to determine how similar two words (more specifically, their embeddings) are. The two simplest ways to do this are:

1. the standard dot product.
$$\text{score}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} \tag{1}$$
2. cosine similarity.
$$\text{score}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||\,||\mathbf{y}||} \tag{2}$$

Now that we have the score defined, the heart of attention comes with the query, key, value model. The general idea is this: for every prediction we want to make (whether it is classifying something or generating

[1] CNNs can be used to encode or output images and RNNs can be used to encode or output sequential data.

a new output word/token), we want to get its respective query vector and use it to look through key-value dictionary in order to get the most relevant information from it. With this information, combined with the query and whatever other information we have, we can make a prediction.

Let's go through this step by step. What we want to do is associate every input hidden node $\mathbf{h}_t$ with a 2-tuple consisting of a key-value pair.

$$\mathbf{h}_t \mapsto (\mathbf{k}_t, \mathbf{v}_t) \tag{3}$$

and associate every hidden output node $\mathbf{s}_t$ with a query value.

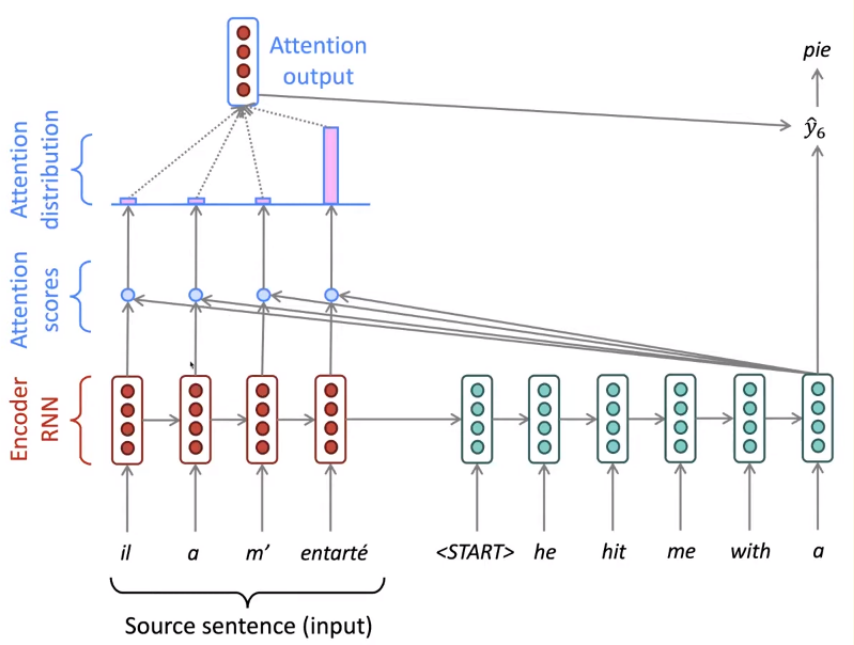
$$\mathbf{s}_T \mapsto \mathbf{q}_t \tag{4}$$

Definition 1.2 (Seq2Seq with Vanilla Attention)

Eventually, we would like to learn these key, value, queries, but for now let's focus on the forward propagation.

1. The input has been sequentially encoded and we have a special start token $\mathbf{s}_0$.
2. For $t' = 0$ until the $\mathbf{s}_{T'}$ is the end token, do the following.
   (a) Take the query $\mathbf{s}_i$ and compute the attention score $\text{score}(\mathbf{s}_i, \mathbf{h}_t)$ for all $t = 1, \dots, T$. This determines which encoder hidden state we should pay attention to.
   $$\mathbf{e}^{t'} = \left[\text{score}(\mathbf{s}_{t'}, \mathbf{h}_1), \dots, \text{score}(\mathbf{s}_{t'}, \mathbf{h}_T)\right] \in \mathbb{R}^T \tag{5}$$
   (b) We take its softmax to get the **attention distribution** $\boldsymbol{\alpha}^{t'}$ for this step (a discrete probability distribution)
   $$\boldsymbol{\alpha}^{t'} = \text{softmax}(\mathbf{e}^{t'}) \in \mathbb{R}^T \tag{6}$$
   (c) We use $\boldsymbol{\alpha}^{t'}$ to take a weighted sum of the encoder hidden states to get the attention output $\mathbf{a}_t$
   $$\mathbf{a}_{t'} = \sum_{t=1}^{T} \alpha_t^{t'} \mathbf{h}_t \in \mathbb{R}^h \tag{7}$$
   which acts as our context vector $\mathbf{C}_{t'}$ that we can now use in our vanilla seq2seq model.[a] Note that this context vector is different for every $\mathbf{s}_{t'}$, so at every step we can choose which encoder states/words to focus on. [b]

Another trick to improve performance is that these attention context vectors can be concatenated with the previous decoder hidden state to get more information in the already decoded part of the sentence.

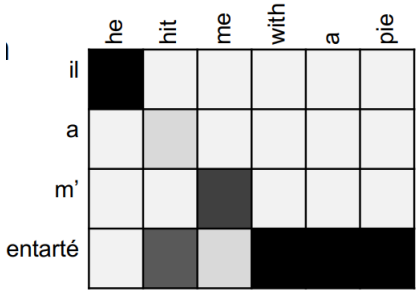


[a]This weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on. Attention is a way to obtain a fixed size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).

[b]This is similar to a hash map where you have a set of key-value pairs. When you have a query, you want to search through the keys to see if it matches the query, and then it returns the value of the matched key. In our case, we have a query, and rather than looking for exact matches, we want to return a similarity score of the query q_i across all k_i's and provide a weighted sum of the corresponding values.

Overall, attention is extremely useful in improving all performance, and it is intuitive with how humans analyze things, too. It significantly improves neural machine translation by allowing the decoder to focus on certain parts of the source. It also provides more "human-like" model of the machine translation process (you can look back at the source sentence while translating, rather than needing to remember it all). It solves the bottleneck problem and helps with the vanishing gradient problem with these pseudo-residual connections through the context vector.

Finally, it provides some interpretability, as we can inspect the attention distribution to see what the decoder was focusing on (which again, we've never set explicitly but was learned by the model).



Code 1.1 ()

For an implementation of this in PyTorch, look here.

Now let's talk about how these parameters are already learned. The parameters of the model are.

1. The encoding matrices for the usual seq2seq model: $\mathbf{W}_e, \mathbf{U}_e$.
2. Generating key, value, and query vectors for every possible embedding is not practical.[2] A more compact way to represent them are through linear maps, so we are learning matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ such that

$$\begin{aligned} \mathbf{q}_{t'} &= \mathbf{Q}\mathbf{s}_{t'} \\ \mathbf{k}_t &= \mathbf{K}\mathbf{h}_t \\ \mathbf{v}_t &= \mathbf{V}\mathbf{h}_t \end{aligned}$$

3. We should now have the decoding matrices $\mathbf{W}_d$ that takes in the attention context vector (plus maybe the previous hidden decoder state) and the original matrix $\mathbf{U}_d$.

Therefore, this problem reduces to learning the matrices $\mathbf{W}_e, \mathbf{U}_e, \mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{W}_d, \mathbf{U}_d$.

Example 1.1 (Score Functions)

Having additional flexibility with the score functions can also improve learning. We provide more examples here, which give more parameters to learn as well.

1. The general attention model allows us to train a shared-weight matrix, allowing for $s \neq h$.

$$e_t^{t'} = \text{score}(\mathbf{s}_{t'}, \mathbf{h}_t) = \mathbf{s}_{t'}^T \mathbf{W}_a \mathbf{h}_t \in \mathbb{R}$$

   However, it may seem like there are too many parameters in $\mathbf{W}_a$, having to learn sh values.
2. The reduced rank multiplicative attention uses low rank matrices, allowing us to learn only $ks + kh$ parameters for matrices $\mathbf{U}_a \in \mathbb{R}^{k \times s}, \mathbf{V}_a \in \mathbb{R}^{k \times h}$ where $k << s, h$.

$$e_t^{t'} = \text{score}(\mathbf{s}_{t'}, \mathbf{h}_t) = \mathbf{s}_{t'}^T (\mathbf{U}_a^T \mathbf{V}_a) \mathbf{h}_t = (\mathbf{U}_a \mathbf{s}_{t'})^T (\mathbf{V}_a \mathbf{h}_t) \in \mathbb{R}$$

3. Additive attention uses a neural net layer defined

$$e_t^{t'} = \mathbf{v}_a^T \tanh(\mathbf{W}_a \mathbf{h}_t + \mathbf{V}_a \mathbf{s}_{t'}) \in \mathbb{R}$$

   where $\mathbf{W}_a \in \mathbb{R}^{r \times h}, \mathbf{V}_a \in \mathbb{R}^{r \times s}$ are weight matrices, $\mathbf{v}_a \in \mathbb{R}^r$ is a weight vector, and r (the attention dimensionality) is a hyperparameter.

This can be naturally extended to other architectures, as we will explore later.

Example 1.2 (Images)

Given an image of size 224×224, we can make patches of size 16×16, and then flatten them to get a 196×768 matrix with a 2-dimensional positional encoding scheme. We can then apply a linear transformation to get the query, key, and value vectors.

[2] You would need three $d_{\text{embedding}} \times |\mathcal{V}|$ matrices, where $\mathcal{V}$ is the set of our vocabulary, which can go easily past 500,000 elements.

2 Transformers

2.1 Self-Attention Layer

While we have solved the bottleneck problem, this entire process is still sequential since every hidden decoder node requires us to know the previous hidden node. There are two sequential processes in the regular seq2seq attention model.

1. The sequential encoding of the input sentence.
2. The sequential decoding of the output sentence. This unfortunately is not possible in transformers to parallelize.

We will focus on parallelizing the first part by temporarily forgetting about encoder-decoder models and just thinking about how to incorporate a good encoder with attention that is parallelizable [VSP+17]. This extension is quite simple. Let $\mathbf{w}_{1:n}$ be a sequence of words in vocabulary $\mathcal{V}$. The key here is that rather than inputs having key-values and outputs having queries, *all* words are associated with a 3-tuple of key, value, query.

$$\mathbf{x}_i \mapsto (\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i) \tag{8}$$

Definition 2.1 (Standard Scaled Dot-Product Attention)

If we keep the score function to be the dot-product, the derivations become quite simple.

1. For each $\mathbf{w}_i$, let $\mathbf{x}_i = E\mathbf{w}_i$ be the word embedding (with $E \in \mathbb{R}^{d \times |\mathcal{V}|}$ the embedding matrix).
2. We transform each word embedding with the (learned) weight matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$.

$$\begin{aligned} \mathbf{q}_i = \mathbf{Q}\mathbf{x}_i &\implies Q = \mathbf{Q}X \\ \mathbf{k}_i = \mathbf{K}\mathbf{x}_i &\implies K = \mathbf{K}X \\ \mathbf{v}_i = \mathbf{V}\mathbf{x}_i &\implies V = \mathbf{V}X \end{aligned}$$

   where $X = [x_1, \dots, x_n] \in \mathbb{R}^{d \times n}$.
3. We compute pairwise similarities between keys and queries and normalize with the softmax to get the attention distribution for each word.

$$\mathbf{e}_{ij} = \mathbf{q}_i^T \mathbf{k}_j, \quad \boldsymbol{\alpha}_{ij} = \frac{\exp(\mathbf{e}_{ij})}{\sum_{j'} \exp(\mathbf{e}_{ij'})} \tag{9}$$

4. Compute the output for each word as a weighted sum of values.

$$\mathbf{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \mathbf{v}_j \tag{10}$$

Ultimately, this can be parallelized into one matrix operation.[a]

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{E_k}}\right)V \tag{11}$$

where the softmax is done to each row.[b][c]
Therefore, when you do a forward pass on an attention layer with input x, you first get the query vector q, extract the attention-weighted values from the key-value dictionary, and then return the weighted sum of the values. To give explicit parameterizations using the query, key, value encoding matrices, we can write this as

$$\text{Attention}(x\,;\,\text{Q}, \text{K}, \text{V}) = \text{softmax}\left(\frac{(\mathbf{Q}x)(\mathbf{K}x)^T}{\sqrt{E_k}}\right)(\mathbf{V}x) \tag{12}$$

This will give us a vector $\mathbf{o}_{1:n}$ consisting of the encoded vectors for each word in the sentence, and best of all, this is parallelizable!

[a]Note that in order to even do such a thing, we must know n beforehand. This can be solved by simply fixing some maximum length, padding everything to be some null token after the end token, and masking all the null tokens to be 0. More on masking later.

[b]We divide by $\sqrt{E_k}$ to stabilize the gradients since as dimensionality increases, the dot product between random vectors tend to get large, leading to large softmax inputs. You can simply compute the variance of two d-dimensional Gaussian vectors and see that their variances scales linearly with d.

[c]You can see that if we have simple dot-product similarity scores, then $E_k = E_v$, but this need not be true in general. We will explore other similiarity score in the next subsection.

There are three problems however.

1. This self-attention encoding does not account for the position/order of the words. Therefore, some positional embedding is needed.
2. Our plan is to stack this layer multiple times on top of each other. However, we are just composing linear maps ultimately, so some nonlinearity is needed.
3. To use self-attention in *decoders*, as we will see later on, we don't want to have any attention on later parts of a sentence, so we need some way to *mask* future words.

We will deal with the first two problems and address the third problem in the transformer archictecture.

2.2 Tokenization and Positional Embeddings

Given an input (or an output) $\mathbf{x}$, it must be tokenized into a sequence of tokens. This is a general preprocessing step that is done for any input, whether it be a sequence of words, a sequence of regions in an image, or a sequence of anything. The raw token data will be denoted w_i for $i = 1, \dots, n$.We can then embed these tokens into a vector space $x_i \in \mathbb{R}^d$.

As we will see later, attention does not have a way to discern the order of the input sequences. Therefore, we must add this positional information to the encoding. The most obvious way would be to simply concatenate the position of the token to the end with an index.

$$x_i \mapsto [x_i, i] \tag{13}$$

However, this is not ideal since this tends to corrupt the embedding of the token. Instead, we can think of adding certain vectors representing components to the original embedding.

$$x_i \mapsto x_i + p_i \tag{14}$$

Certain ways come to mind, such as simply letting p_i be the vector of all i's. This tends not to work in progress since the values of i get too large and corrupts the embeddings too much.[3] Normalizing the values of i to be in $[0, 1]$ is disadvantageous because now the positional embedding p_i is dependent on the length of the total input sequence. Therefore, we need two properties:

1. The positional encoding should be independent of the input sequence length.
2. The positional encoding shouldn't be too large that it corrupts the semantic meaning behind the original embedding.

It turns out that the sinusoidal function satisfies these properties.

Definition 2.2 (Sinusoidal Position Embedding)

Given the embeddings $x_i \in \mathbb{R}^d$, we can add a positional encoding to it by

$$x_i \mapsto x_i + p_i$$

[3]A helpful Medium article here

where the positional encoding is given by the vector where each component is defined as

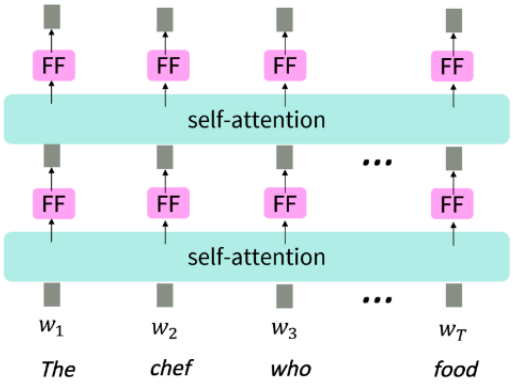
$$(p_i)_j = \begin{cases} \sin\left(\frac{i}{10000^{2j/d}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{2j/d}}\right) & \text{if } j \text{ is odd} \end{cases} \tag{15}$$

where i iterates through the tokens and j iterates through the dimensions of the embedding.

2.3 Stacked Attention Layers and Multi-Head Attention

The second problem of introducing nonlinearities is quite simple. Once we have the output of the first self-attention layer $\mathbf{o}_{1:n} = [\mathbf{o}_1, \dots, \mathbf{o}_n]$, we can just input each $\mathbf{o}_i$ through a small MLP to introduce nonlinearity before inputting it into the next self-attention layer.

$$\begin{aligned} m_i &= MLP(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2 \end{aligned}$$



Boom, problem solved.

Going back, if we want to look at the attention for token x_i, we want to look through all $q_i^T k_j$ for all j and find out where it is high. But perhaps we want to focus on different j for different reasons. The following example may illustrate why.

Example 2.1 (Semantic and Syntactic Attention)

Given the sentence *I went to the bank and got some money.*, one type of attention may look at the semantic meaning of the words, such as associating *bank* with *money*. However, we may also want to look at the syntactic meaning of the words, such as associating *went* with *bank*. When we read sentences, we have different types of attention for different reasons, and so having multiple heads of attention may be useful.

Definition 2.3 (Multi-Head Attention)

Therefore, let us construct multiple attention heads by defining multiple triplets of (Q, K, V) matrices. This may be more computationally inefficient, so we simply scale down the size of these matrices from

$$Q \in \mathbb{R}^{E_q \times d}, K \in \mathbb{R}^{E_k \times d}, V \in \mathbb{R}^{E_v \times d}$$

to

$$Q_\ell \in \mathbb{R}^{E_q \times d/h}, K_\ell \in \mathbb{R}^{E_k \times d/h}, V_\ell \in \mathbb{R}^{E_v \times d/h}$$

where h is the number of heads. We are essentially decreasing the size of the token embedding dimension in order to get more heads. We can then do attention on each head separately.

$$\text{Attention}_\ell = \text{Attention}(Q_\ell, K_\ell, V_\ell) = \text{softmax}\left(\frac{Q_\ell K_\ell^T}{\sqrt{E_k/h}}\right)V_\ell \tag{16}$$

and we can simply concatenate them together to get the final output.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{Attention}_1, \dots, \text{Attention}_h)W^O$$

where $W^O \in \mathbb{R}^{d \times E_v}$ is a learnable weight matrix that mixes these heads together with a final linear transformation.[a] This entire process is shown in Figure 1.

[a] There is a valid concern that these heads may all end up learning the same thing and may just converge onto the same thing. However, this is not what happens in practice.

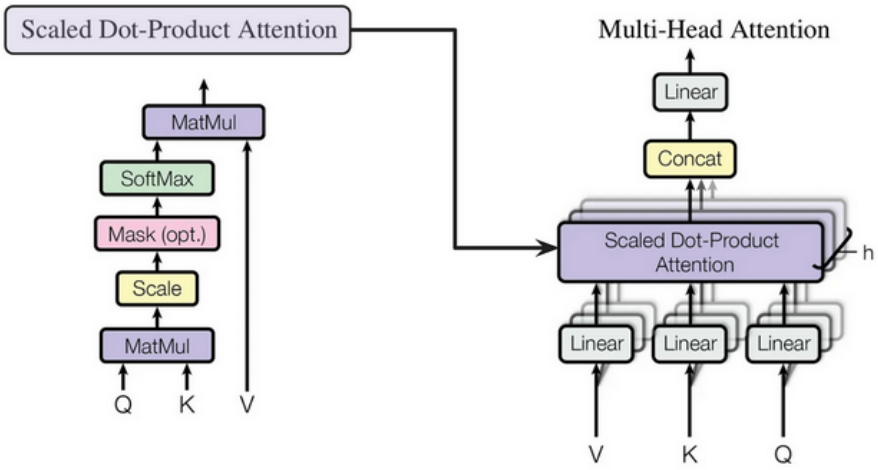


Figure 1: Diagram of multi head attention.

With self-attention out of the way, the transformer architecture becomes quite simple. An overview of it is shown in Figure 2.

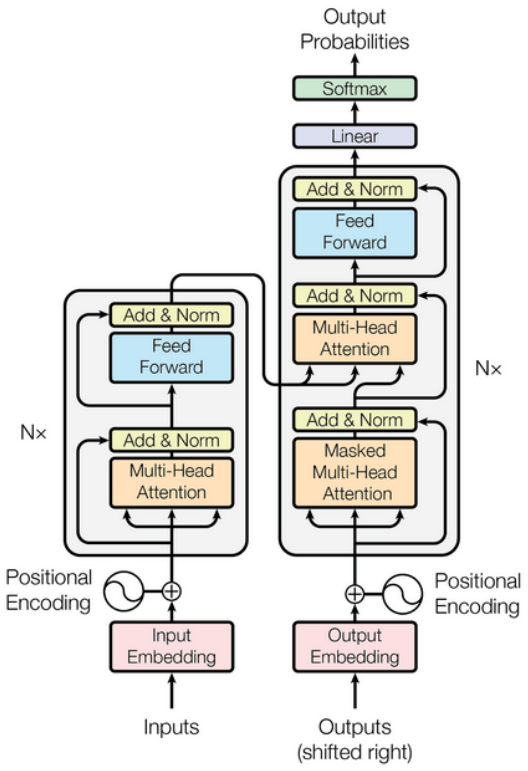


Figure 2: Transformer architecture.

The encoder is quite simple. You take the input embedding and add the positional embeddings to get $\{x_i \in \mathbb{R}^d\}_{i=1}^n$. You then pass it through a multi-head self-attention layer, which has outputs of shape $E_v \times n$, and then pass it through a feed forward network, adding residual connections and normalization layers to help with training. You then repeat this process N times, which gives out the encoded sequence $\mathbf{h}_{1:n} = [\mathbf{h}_1, \dots, \mathbf{h}_n]$, now ready to be fed into the decoder.[4]

The decoder has two different self-attention layers. First, we run the generated output sequence through a masked self-attention layer, which generates the hidden nodes $\mathbf{z}_{1:n} = [\mathbf{z}_1, \dots, \mathbf{z}_n]$ representing the state of the currently decoded sentence. Again, we have some maximum output length to ensure that we are working with a fixed size, and manually mask all tokens after the current one to be 0.

Then, another **cross-attention** layer takes both $\mathbf{h}_{1:n}$ and $\mathbf{z}_{1:n}$ and with its trained $(\mathbf{K}, \mathbf{V}, \mathbf{Q})$, computes the key, value, and query matrices as

$$K = \mathbf{K}\mathbf{h}_{1:n}, \quad V = \mathbf{V}\mathbf{h}_{1:n}, \quad Q = \mathbf{Q}\mathbf{z}_{1:n}, \tag{17}$$

now ready to be plugged into to the self-attention formulas, integrating both the inputs and the current output to generate the result. This again outputs another list of vectors, which are run through an MLP and then have another set of $(\mathbf{K}, \mathbf{V}, \mathbf{Q})$ matrices waiting for them. This makes sense, since we want to use the output sequence to query the input key-values and attend to the correct set of tokens.

The output of this is then passed through a feed forward network, with some residual connections and normalization, and finally a linear layer transforms the output dimensions to whatever is needed (e.g. size of the vocabulary, or number of classes). Once this is done, a new word is generated,[5] and this word (along with all previous words) is now used as the new input to the decoder in place of the start token. This process

[4] You can see that to support iterating through n times, E_v should equal d.

[5] Note that we have not specified how to get the corresponding word given an embedding vector. This is not within the scope of these notes and are covered in my natural language processing notes.

is done until the stop token is generated by the decoder. Notice that we encode with a bidirectional model (no masking) and generated the target with a unidirectional model (masking).

Note that again, parallelization of the decoder is not possible in the transformer architecture. Additionally, you can see that more normalization layers and residual connections are needed to train efficiently. This is very important in practice.

Despite all its advantages, self-attention has quadratic runtime complexity with respect to the sequence length since we need to compute attention for all pairs of words. This is worse than the linear runtime complexity in RNNs.

2.3.1 Masking

The final aspect we did not address is the masking. When we are training the transformer on a corpus of data, the decoder first computes self-attention on all the previous outputs first to get the query, and then takes in the output of the encoder self-attention layer as the keys and values. Then it does self-attention once more over these triplets, essentially doing a self-attention layer over the entire input and all tokens up to the current decoded output.

When training this model, we have access to the entire decoded output, and we want to make sure that we do not perform self-attention on any future words since it will most likely attend 100% to the next word to generate the next word! This does not learn anything, so we artificially set the attention distribution for all future output words to be 0. This is usually done by setting the attention scores to $-\infty$ (or more practically, a very negative number) which will result in 0 after softmaxing.[6]

2.4 Practical Implementation

In here we go over the nitty gritty details that comes into implementing a transformer in `pytorch==2.3.0`.

2.4.1 Key, Value, Query Matrices and In Projections

The first thing is that these key, value, query does not have to necessarily equal to the dimension embedding, which we will denote as E. One flexibility is that we don't necessarily need to set the dimensions of the keys, values, and queries the same. We can see in the constructor of the `torch.nn.MultiheadAttention` module that you can input your own dimensions for the keys and values, but queries must be the same as E.

```
def __init__(self, embed_dim, num_heads, dropout=0., bias=True, add_bias_kv=False,
    add_zero_attn=False,
             kdim=None, vdim=None, batch_first=False, device=None, dtype=None) -> None:
   ...
  self.embed_dim = embed_dim
  self.kdim = kdim if kdim is not None else embed_dim
  self.vdim = vdim if vdim is not None else embed_dim
  self._qkv_same_embed_dim = self.kdim == embed_dim and self.vdim == embed_dim
```

In fact, $K \in \mathbb{R}^{d_k \times E}$, $V \in \mathbb{R}^{d_v \times E}$, then $QK^T \in \mathbb{R}^{E \times d_k}$. Since this obviously leads to dimension mismatch problem when we multiply it with the matrix V, what we do is have an **in projection** layer that maps everything to dimension E. We can check this for the following.

$$K_{proj} \in \mathbb{R}^{E \times d_k}, V_{proj} \in \mathbb{R}^{E \times d_v}, Q_{proj} \in \mathbb{R}^{E \times E} \tag{18}$$

There are two ways to store these projection matrices, as shown in the constructor.

```
# in the constructor
...
self._qkv_same_embed_dim = self.kdim == embed_dim and self.vdim == embed_dim
```

[6] Here is a nice explanation here.

```
4
5  if not self._qkv_same_embed_dim:
6    self.q_proj_weight = Parameter(torch.empty((embed_dim, embed_dim), **factory_kwargs))
7    self.k_proj_weight = Parameter(torch.empty((embed_dim, self.kdim), **factory_kwargs))
8    self.v_proj_weight = Parameter(torch.empty((embed_dim, self.vdim), **factory_kwargs))
9    self.register_parameter('in_proj_weight', None)
10 else:
11   self.in_proj_weight = Parameter(torch.empty((3 * embed_dim, embed_dim), **factory_kwargs))
12   self.register_parameter('q_proj_weight', None)
13   self.register_parameter('k_proj_weight', None)
14   self.register_parameter('v_proj_weight', None)
```

1. If these shapes are different, then we store them in separate matrices as above.

```
att = nn.MultiheadAttention(embed_dim=50, num_heads=1, bias=False, kdim=30, vdim=40)
att.q_proj_weight.shape # torch.Size([50, 50])
att.k_proj_weight.shape # torch.Size([50, 30])
att.v_proj_weight.shape # torch.Size([50, 40])
```

2. If these shapes are the same, then we just store them in a $3E \times E$ matrix by concatenation them.

```
att = nn.MultiheadAttention(embed_dim=50, num_heads=1, bias=False)
att.in_proj_weight.shape # torch.Size([150, 50])
```

These conditions are asserted throughout the forward pass as well.

2.4.2 Masking

We multiply by a masking matrix.

2.4.3 Computing Attention

First we reshape them so that they are batch first.

If `needs_weights = True`, we also output the attention weights in addition to the output, but it is said that this degrades performance. It is by default true but should be set to false for small tasks.

2.4.4 Forward Pass of MultiheadAttention

First, we should look at the main function that computes self-attention. We omit a large part of the code to focus on the relevant details.

```
# torch.nn.functional
def multi_head_attention_forward(
  query: Tensor,
  key: Tensor,
  value: Tensor,
  embed_dim_to_check: int,
  num_heads: int,
  in_proj_weight: Optional[Tensor],
  in_proj_bias: Optional[Tensor],
  bias_k: Optional[Tensor],
  bias_v: Optional[Tensor],
  add_zero_attn: bool,
  dropout_p: float,
  out_proj_weight: Tensor,
  out_proj_bias: Optional[Tensor],
```

```
  training: bool = True,
  ...
):
# first unsqueezes the input if it is not batched.

# look at the input dimensions and check that multiheads divide it evenly
#
assert embed_dim == embed_dim_to_check, \
    f"was expecting embedding dimension of {embed_dim_to_check}, but got {embed_dim}"
if isinstance(embed_dim, torch.Tensor):
    # embed_dim can be a tensor when JIT tracing
    head_dim = embed_dim.div(num_heads, rounding_mode='trunc')
else:
    head_dim = embed_dim // num_heads
assert head_dim * num_heads == embed_dim, f"embed_dim {embed_dim} not divisible by num_heads
    {num_heads}"
if use_separate_proj_weight:
    # allow MHA to have different embedding dimensions when separate projection weights are
    used
    assert key.shape[:2] == value.shape[:2], \
        f"key's sequence and batch dims {key.shape[:2]} do not match value's {value.shape[:2]}"
else:
    assert key.shape == value.shape, f"key shape {key.shape} does not match value shape
    {value.shape}"

# Computes in-projection, which is an affine map before doing attention.
# in_proj_weight = [W_q, W_k, W_v], in_proj_bias = [b_q, b_k, b_v]
# computes q = q * W_q + b_q, k = k * W_k + b_k, v = v * W_v + b_v
if not use_separate_proj_weight:
  q, k, v = _in_projection_packed(query, key, value, in_proj_weight, in_proj_bias)
else:
  if in_proj_bias is None:
      b_q = b_k = b_v = None
  else:
      b_q, b_k, b_v = in_proj_bias.chunk(3)
  q, k, v = _in_projection(query, key, value, q_proj_weight, k_proj_weight, v_proj_weight,
    b_q, b_k, b_v)

# prepare attention mask
# add bias along batch dimension
# more preparation with mask
...
# Now calculate attention
if need_weights:
  # scale q_scale for the sqrt(E) division factor
  B, Nt, E = q.shape
  q_scaled = q * math.sqrt(1.0 / float(E))

if attn_mask is not None:
  # torch.baddbmm is a pybinded C function implementing matrix multiplication
  # of form attn_mask + q_scaled @ k^T
  attn_output_weights = torch.baddbmm(attn_mask, q_scaled, k.transpose(-2, -1))
else:
  # torch.bmm is also a pybinded C function q_scaled + k^T
  attn_output_weights = torch.bmm(q_scaled, k.transpose(-2, -1))
...
# softmax it and then multiply it by V.
attn_output_weights = softmax(attn_output_weights, dim=-1)
attn_output = torch.bmm(attn_output_weights, v)
```

```

    # final linear layer for more weightings.
    attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len * bsz, embed_dim)
    attn_output = linear(attn_output, out_proj_weight, out_proj_bias)
    attn_output = attn_output.view(tgt_len, bsz, attn_output.size(1))

    # optionally average attention weights over heads
    attn_output_weights = attn_output_weights.view(bsz, num_heads, tgt_len, src_len)
    if average_attn_weights:
        attn_output_weights = attn_output_weights.mean(dim=1)

    if not is_batched:
        # squeeze the output if input was unbatched
        attn_output = attn_output.squeeze(1)
        attn_output_weights = attn_output_weights.squeeze(0)
    return attn_output, attn_output_weights
```

This is precisely the function that is called in the forward method of the `MultiheadAttention` module.

2.4.5 Transformer

In the transformer, we can see that if we peek at the state dictionary, it composes of an encoder and a decoder, each with a certain number of attention layers. There are 6 attention layers each by default.

```
transformer = nn.Transformer()
transformer.state_dict
# output
<bound method Module.state_dict of Transformer(
  (encoder): TransformerEncoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_feat
ures=512, out_features=512, bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048,
 bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512,
 bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=T
rue)
  )
  (decoder): TransformerDecoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerDecoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_feat
ures=512, out_features=512, bias=True)
        )
```

```
        (multihead_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_feat
ures=512, out_features=512, bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048,
 bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512,
 bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (norm3): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
        (dropout3): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=T
rue)
  )
)>
```

3 Vision Transformers

We have hinted at attention being applicable in other architectures, and the most popular is in computer vision. Historically, CNNs were very useful because they take into account the locality and translational-invariance of objects in images inherently in the convolutions. This is a great strength of convolutional networks.

Can transformers beat this? These assumptions are not built into the architecture, and researchers were quite unsuccessful in passing the benchmarks set by CNNs, but it turned out that in 2020, with enough training and a large enough architecture, *vision transformers* in fact did surpass CNNs.

References

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.