EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

Homework 9

This homework is due on Apr 27, at 10:59PM.

1. Justifying Scaled-Dot Product Attention

Suppose $q, k \in \mathbb{R}^d$ are two random vectors with $q, k \overset{iid}{\sim} N(\mu, \sigma^2 I)$, where $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^+$. In other words, each component q_i of q is drawn from a normal distribution with mean μ and standard deviation σ, and the same is true for k.

(a) Define $\mathbb{E}[q^T k]$ in terms of μ, σ and d.

(b) Considering a practical case where $\mu = 0$ and $\sigma = 1$, define $\text{Var}(q^T k)$ in terms of d.

(c) Continue to use the setting in part (b), where $\mu = 0$ and $\sigma = 1$. Let s be the scaling factor on the dot product. Suppose we want $\mathbb{E}[\frac{q^T k}{s}]$ to be 0, and $\text{Var}(\frac{q^T k}{s})$ to be $\sigma = 1$. What should s be in terms of d?

2. Argmax Attention

Recall from lecture that we can think about attention as being *queryable softmax pooling*. In this problem, we ask you to consider a hypothetical argmax version of attention where it returns exactly the value corresponding to the key that is most similar to the query, where similarity is measured using the traditional inner-product.

(a) Perform **argmax attention** with the following keys and values:

Keys:

$$\left\{ \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}, \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

Corresponding Values:

$$\left\{ \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \right\}$$

using the following query:

$$\mathbf{q} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

What would be the output of the attention layer for this query?

Hint: For example, $\text{argmax}([1, 3, 2]) = [0, 1, 0]$

(b) Note that instead of using *softmax* we used `argmax` to generate outputs from the attention layer. **How does this design choice affect our ability to usefully train models involving attention?**
(Hint: think about how the gradients flow through the network in the backward pass. Can we learn to improve our queries or keys during the training process?)

3. Ordinary Softmax Multihead Attention Implementation

(a) Complete the incomplete lines in the implementation of multi-head attention (with a final linear layer on top) by filling in (on the following page) what (a),(b),(c), etc. should be in the code below.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, ___(a)___)
        self.W_v = nn.Linear(___(b)___, d_model)
        self.W_o = nn.Linear(d_model, ___(c)___)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(___(d)___, K.transpose(-2, -1))
        attn_scores = attn_scores /  math.sqrt(self.d_k)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
        attn_probs = torch.___(e)____(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, ____(f)___)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length,
                      _____(g)_____, self.d_k).transpose(1, 2)

    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).view(batch_size,
                                      seq_length, self.d_model)

    def forward(self, Q, K, V, mask=None):
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))
```



```
        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        output = self._____(h)_____(self.combine_heads(attn_output))
        return output
```

Fill in the missing blanks. Some have been given to you so you can see what kind of answers we are looking for. You need to write in the boxes.

(a)

(b)

(c) `d_model`

(d)

(Hint: which of `Q`*,* `K`*,* `V` *should go here?)*

(e)

(Hint: What operation do we need to use to go from scores to probabilities?)

(f)

(Hint: which of `Q`*,* `K`*,* `V` *should go here?)*

(g) `self.num_heads`

(h)

(Hint: which linear model do we still need to apply?)

(b) Suppose that you want to update the initialization `W_v` in this way:

```
self.W_v = nn.Linear(in_features=d_model, out_features=2 * d_model)
```

This increases the length of the value vectors that we get.

What other minimal modifications should you apply to the code to make the code run?

(Hint: Remember that in transformers, we typically use attention blocks with residual connections around them. What does that require?)

4. Transformer Decoding Optimization

Consider a multi-head attention (MHA) layer in a transformer decoder performing incremental decoding. The attention layer operates on a batch of b sequences, each with a dimension of d. For each of the h heads, the dimensions of query, key, and value projections are $q = \frac{d}{h}$, $k = \frac{d}{h}$, and $v = \frac{d}{h}$ respectively.

The following pseudocode performs the multi-head self-attention operation for a single token generation in the sequence (for each sequence in our batch).

```
def multihead_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v, W_o):
    """Multi-head Self-Attention (one step).
    Args:
        x: a tensor with shape [b, d] - current token embedding
        prev_K: tensor with shape [b, h, n, k] - cached keys from prev. tokens
        prev_V: tensor with shape [b, h, n, v] - cached values from prev.
            tokens
```



```
        W_q: a tensor with shape [h, d, q] - query projection weights
        W_k: a tensor with shape [h, d, k] - key projection weights
        W_v: a tensor with shape [h, d, v] - value projection weights
        W_o: a tensor with shape [h, v, d] - output projection weights
    Returns:
        y: a tensor with shape [b, d] - output embedding
    """
    # Project inputs to query, key, and value
    q = torch.einsum("bd,hdq->bhq", x, W_q)  # [b, h, q]
    k = torch.einsum("bd,hdk->bhk", x, W_k)  # [b, h, k]
    v = torch.einsum("bd,hdv->bhv", x, W_v)  # [b, h, v]

    # Append new key and value to previous cached states
    new_K = torch.cat([prev_K, k.unsqueeze(2)], dim=2)  # [b, h, n+1, q]
    new_V = torch.cat([prev_V, v.unsqueeze(2)], dim=2)  # [b, h, n+1, v]

    # Compute attention scores, apply softmax, and get weighted values
    logits = torch.einsum("bhq,bhnq->bhn", q, new_K)  # [b, h, n+1]
    weights = torch.nn.functional.softmax(logits, dim=-1)  # [b, h, n+1]
    o = torch.einsum("bhn,bhnv->bhv", weights, new_V)  # [b, h, v]

    # Project back to output dimension
    y = torch.einsum("bhv,hvd->bd", o, W_o)  # [b, d]

    return y
```

Multi-Query Attention (MQA): Multi-query attention is a variant of multi-head attention where all heads share the same keys and values, reducing memory access and improving efficiency for incremental decoding.

```
def multiquery_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v, W_o):
    """Multi-query Self-Attention (one step).
    Args:
        x: a tensor with shape [b, d] - current token embedding
        prev_K: tensor with shape [b, n, k] - cached keys (shared across heads
            )
        prev_V: tensor with shape [b, n, v] - cached values (shared across
            heads)
        W_q: a tensor with shape _____A____ - query projection weights
        W_k: a tensor with shape _____B____ - key projection weights
        W_v: a tensor with shape [d, v] - value projection weights
        W_o: a tensor with shape [h, v, d] - output projection weights
    Returns:
        y: a tensor with shape [b, d] - output embedding
    """
    # Project input to queries (per head), keys and values (shared)
    q = _____C_____
    k = _____D_____
    v = _____E_____

    # Append new key and value to previous cached states
    new_K = torch.cat([prev_K, k.unsqueeze(1)], dim=1)  # [b, n+1, k]
    new_V = torch.cat([prev_V, v.unsqueeze(1)], dim=1)  # [b, n+1, v]

```



```
    # Compute attention scores, apply softmax, and get weighted values
    logits = torch.einsum("bhk,bnk->bhn", q, new_K)      # [b, h, n+1]
    weights = torch.nn.functional.softmax(logits, dim=-1)  # [b, h, n+1]
    o = torch.einsum("bhn,bnv->bhv", weights, new_V)     # [b, h, v]

    # Project back to output dimension
    y = torch.einsum("bhv,hvd->bd", o, W_o)              # [b, d]

    return y
```

First, fill in the blanks in the code/comments above by picking one:

(a) (3pts) **What is A?**

◯ $[h, d, q]$ ◯ $[d, q]$ ◯ $[h, q]$

(b) (3pts) **What is B?**

◯ $[h, d, k]$ ◯ $[d, k]$ ◯ $[h, k]$

(c) (4pts) **What is C?**

◯ `torch.einsum("bd,dq->bhq", x, W_q)`

◯ `torch.einsum("bd,hdq->bhq", x, W_q)`

◯ `torch.einsum("bhd,dq->bhq", x, W_q)`

(d) (4pts) **What is D?**

◯ `torch.einsum("bd,dk->bk", x, W_k)`

◯ `torch.einsum("bd,hdk->bhk", x, W_k)`

◯ `torch.einsum("bd,dk->bhk", x, W_k)`

(e) (4pts) **What is E?**

◯ `torch.einsum("bd,hdv->bhv", x, W_v)`

◯ `torch.einsum("bd,dv->bhv", x, W_v)`

◯ `torch.einsum("bd,dv->bv", x, W_v)`

(f) (4pts) Recall that for generating a batch of single tokens in multi-head-attention with a cache of n previous tokens, the operations have computational complexity $O(bd^2 + bnd)$ since it is $O(bd^2)$ to do the projections (query, key, value, output) and $O(bnd)$ from attention (dot products with n cached keys/values for each head). The memory access complexity is $O(d^2 + bd + bnd)$ since it is $O(d^2)$ to read weights, $O(bd)$ to read/write current token embeddings (x, q, k, v, o, y), and $O(bnd)$ to read cached keys/values. The MHA arithmetic intensity is thus $\frac{O(bd^2+bnd)}{O(d^2+bd+bnd)}$.

- Both terms are kept; which one dominates depends on the relative size of n and d.
- If n is small, projections dominate; if n is large, memory access for cached keys/values dominates.

Now consider MQA. For generating a single token with a cache of n previous tokens, **what is the computational complexity (in Big-O notation)?**

(g) (4pts) For MQA generating a single token with a cache of n previous tokens, **what is the memory access complexity (in Big-O notation)?**

5. Coding Question: Visualizing Attention

Please run the cells in the Visualizing_BERT.ipynb notebook, then answer the questions below.

(a) Attention in GPT: Run part a of the notebook and generate the corresponding visualizations

i. **What similarities and differences do you notice in the visualizations between the examples in this part?** Explore the queries, keys, and values to identify any interesting patterns associated with the attention mechanism.

ii. **How does attention differ between the different layers of the GPT model? Do you notice that the tokens are attending to different tokens as we go through the layers of the network?**

(b) BERT pays attention: Run part b of the notebook and generate the corresponding visualizations.

i. Look at different layers of the BERT model in the visualizations of part (b) and identify different patterns associated with the attention mechanism. Explore the queries, keys, and values to further inform your answer. **For instance, do you notice that any particular type of tokens are attended to at a given timestep?**

ii. **Do you spot any differences between how attention works in GPT vs. BERT? Think about how the model architectures are different.**

iii. For the example with syntactically similar but definitionally different sentences, look through the different layers of the two BERT networks associated with sentence a and sentence b, and take a look at the queries, keys, and values associated with the different tokens. **Do you notice any differences in the embeddings learned for the two sentences that are essentially identical in structure but different in meaning?**

iv. **For the pre-training related examples, do you notice BERT's bi-directionality in play? Do you think pre-training the BERT helped it learn better representations?**

(c) BERT has multiple heads!: Run part c of the notebook and generate the corresponding visualizations.

i. **Do you notice different features being learned throughout the different attention heads of BERT? Why do you think this might be?**

ii. **Can you identify any of the different features that the different attention heads are focusing on?**

(d) Visualizing untrained attention weights

i. **What differences do you notice in the attention patterns between the randomly initialized and trained BERT models?**

ii. **What are some words or tokens that you would expect strong attention between? What might you guess about the gradients of this attention head for those words?**

(e) **Were you able to identify interesting patterns in the visualizations?** If yes, please share some examples (describe in text or paste a screenshot). If not, feel free to use this space for your frustrations.

6. Kernelized Linear Attention (Part I)

The softmax attention is widely adopted in transformers, however the $\mathcal{O}\left(N^2\right)$ (N stands for the sequence length) complexity in memory and computation often makes it less desirable for processing long document like a book or a passage, where the N could be beyond thousands. There is a large body of the research studying how to resolve this [1].

Under this context, this question presents a formulation of attention via the lens of the kernel. A large portion of the context is adopted from [2]. In particular, attention can be seen as applying a kernel over the inputs

[1] https://huggingface.co/blog/long-range-transformers

[2] Tsai, Yao-Hung Hubert, et al. "Transformer dissection: a unified understanding of transformer's attention via the lens of kernel" (2019).

with the kernel scores being the similarities between inputs. This formulation sheds light on individual components of the transformer's attention, and helps introduce some alternative attention mechanisms that replaces the "softmax" with linearized kernel functions, thus reducing the $\mathcal{O}\left(N^2\right)$ complexity in memory and computation.

We first review the building block in the transformer. Let $x \in \mathbb{R}^{N\times F}$ denote a sequence of N feature vectors of dimensions F. A transformer[3] is a function $T : \mathbb{R}^{N\times F} \to \mathbb{R}^{N\times F}$ defined by the composition of L transformer layers $T_1(\cdot), \ldots, T_L(\cdot)$ as follows,

$$T_l(x) = f_l(A_l(x) + x). \tag{1}$$

The function $f_l(\cdot)$ transforms each feature independently of the others and is usually implemented with a small two-layer feedforward network. $A_l(\cdot)$ is the self attention function and is the only part of the transformer that acts across sequences.

We now focus on the the self attention module which involves softmax. The self attention function $A_l(\cdot)$ computes, for every position, a weighted average of the feature representations of all other positions with a weight proportional to a similarity score between the representations. Formally, the input sequence x is projected by three matrices $W_Q \in \mathbb{R}^{F\times D}$, $W_K \in \mathbb{R}^{F\times D}$ and $W_V \in \mathbb{R}^{F\times M}$ to corresponding representations Q, K and V. The output for all positions, $A_l(x) = V'$, is computed as follows,

$$\begin{aligned} Q = xW_Q, K = xW_K, V = xW_V, \\ A_l(x) = V' = \text{softmax}(\frac{QK^T}{\sqrt{D}})V. \end{aligned} \tag{2}$$

Note that in the previous equation, the softmax function is applied rowwise to QK^T. Following common terminology, the Q, K and V are referred to as the "queries", "keys" and "values" respectively.

Equation 2 implements a specific form of self-attention called softmax attention where the similarity score is the exponential of the dot product between a query and a key. Given that subscripting a matrix with i returns the i-th row as a vector, we can write a generalized attention equation for any similarity function as follows,

$$V_i' = \frac{\sum_{j=1}^{N} \text{sim}\left(Q_i, K_j\right) V_j}{\sum_{j=1}^{N} \text{sim}\left(Q_i, K_j\right)}. \tag{3}$$

Equation 3 is equivalent to equation 2 if we substitute the similarity function with $\text{sim}_{\text{softmax}}(q, k) = \exp(\frac{q^T k}{\sqrt{D}})$. This can lead to

$$V_i' = \frac{\sum_{j=1}^{N} \exp(\frac{Q_i^T K_j}{\sqrt{D}})V_j}{\sum_{j=1}^{N} \exp(\frac{Q_i^T K_j}{\sqrt{D}})}. \tag{4}$$

For computing the resulting self-attended feature $A_l(x) = V'$, we need to compute all V_i' $i \in 1, ..., N$ in equation 4.

(a) **Identify the conditions that needs to be met by the** sim **function to ensure that** V_i **in Equation 3 remains finite (the denominator never reaches zero).**

(b) The definition of attention in equation 3 is generic and can be used to define several other attention implementations.

[3] Vaswani, Ashish, et al. "Attention is all you need" (2017).

(i) One potential attention variant is the "polynomial kernel attention", where the similarity function as sim (q, k) is measured by polynomial kernel $\mathcal{K}$ [4]. **Considering a special case for a "quadratic kernel attention" that the degree of "polynomial kernel attention" is set to be 2, derive the sim (q, k) for "quadratic kernel attention". (NOTE: any constant factor is set to be 1.)** .

(ii) One benefit of using kernelized attention is that we can represent a kernel using a feature map $\phi(\cdot)$ [5]. **Derive the corresponding feature map $\phi(\cdot)$ for the quadratic kernel.**

(iii) **Considering a general kernel attention, where the kernel can be represented using feature map that $\mathcal{K}(q, k) = (\phi(q)^T \phi(k))$, rewrite kernel attention of equation 3 with feature map $\phi(\cdot)$.**

(c) We can rewrite the softmax attention in terms of equation 3 as equation 4. **For all the V_i' ($i \in \{1, ..., N\}$), derive the time complexity (asymptotic computational cost) and space complexity (asymptotic memory requirement) of the above softmax attention in terms of sequence length N, D and M.**

NOTE: for memory requirement, we need to store any intermediate results for backpropagation, including all Q, K, V

(d) Assume we have a kernel $\mathcal{K}$ as the similarity function and the kernel can be represented with a feature map $\phi(\cdot)$, we can rewrite equation 3 with $\text{sim}(x, y) = \mathcal{K}(x, y) = (\phi(Q_i)^T \phi(K_j))$ in part (b). We can then further simplify it by making use of the associative property of matrix multiplication to

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j)}. \tag{5}$$

Note that the feature map $\phi(\cdot)$ is applied row-wise to the matrices Q and K.

Considering using a linearized polynomial kernel $\phi(x)$ of degree 2, and assume $M \approx D$, derive the computational cost and memory requirement of this kernel attention as in (5).

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Sheng Shen.
- David M. Chan.

[4] https://en.wikipedia.org/wiki/Polynomial_kernel
[5] https://en.wikipedia.org/wiki/Kernel_method

- Saagar Sanghavi.
- Yossi Gandelsman.
- Naman Jain.
- Dhruv Shah.
- Shivam Singhal.
- Kevin Li.
- Olivia Watkins.
- Bryan Wu.
- Anant Sahai.
- Shaojie Bai.
- Angelos Katharopoulos.
- Hao Peng.

