EECS 182 Deep Neural Networks
Fall 2026 Gireeja Ranade

Homework 10

This homework is due on Wednesday, November 11, 2026, at 10:59PM.

1. Ridge-Attention

In lecture, you saw how the standard softmax-attention mechanism can be viewed as a softened version of something akin to a nearest-neighbor model in which the value returned for a query reflects a weighted combination of the values that correspond to the keys closest to the query. In this view, the (key, value) pairs in the memory represent a kind of in-context "training data" and the query is a test input for which we want to predict the right output given that data.

(a) To start, let's think about why it is possible to efficiently update simple averaging. Let $m = \frac{1}{n}\sum_{i=1}^{n} x_i$ be the average of n points. **Use m, x_{n+1}, n and simple arithmetic operations to compute $m' = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i$ — the average of all points including the new point x_{n+1}.**
(HINT: Start by multiplying m by n.)

Solution: This question is a warmup. By multiplying m with n, we get that $mn = \sum_{i=1}^{n} x_i$, which means that $\sum_{i=1}^{n+1} x_i = mn + x_{n+1}$ and so $m' = \frac{mn+x_{n+1}}{n+1}$. In other words, we can update a running average with a new point in constant time. We don't have to start from scratch.

(b) Let us now shift to thinking about traditional ridge-regression with n training pairs $(\mathbf{x}_i, y_i)$ where $\mathbf{x}_i$ are d-dimensional vectors and y_i are scalars. Let the matrix $A = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$ and vector $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ so that we can find the familiar closed-form solution

$$\mathbf{w}_* = (A^\top A + \lambda I)^{-1} A^\top \mathbf{y} \tag{1}$$

that allows us to make scalar predictions on any new test input $\mathbf{x}$ by computing $\mathbf{w}_*^\top \mathbf{x}$.
First, **write the two terms $(A^\top A + \lambda I)$ and $A^\top \mathbf{y}$ as sums involving the $\mathbf{x}_i$ and y_i.**
i.e. Complete:
$(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^{n}$

Solution: Linear regression and ridge-regression are just fancy ways to average. It is all about being able to see this clearly with math.
Using the outer-product form of matrix multiplication, since the columns of $A^\top$ are $\mathbf{x}_i$ and the rows of A are $\mathbf{x}_i^\top$, we know that $A^\top A = \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top$. From this we conclude that $(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top$.
This will be useful later because it tells us that adding-in one more data point in ridge regression is tantamount to performing a rank-1 update on the matrix we need to invert.
$A^\top \mathbf{y} = \sum_{i=1}^{n}$

Solution: Since the columns of $A^\top$ are $\mathbf{x}_i$ and matrix-vector multiplication consists of taking a linear combination of the columns of the matrix, we know that $A^\top \mathbf{y} = \sum_{i=1}^{n} \mathbf{x}_i y_i$.

 All Rights Reserved. This may not be publicly shared without explicit permission.

This will be useful later because it tells us that adding one more data point in scalar-output ridge regression is tantamount to adding one more vector in the second part of (1).

(c) Suppose we wanted to do ridge-self-attention (non-causal – "encoder-style") with a context length of n and d-dimensional query, key, and value vectors. Recall that this is the style of attention where each of the n queries is applied to the same pool of n (key, value) pairs. The goal is to calculate multi-dimensional ridge-regression predictions, after "training" on the pool of (key, value) pairs, and given the query as a kind of "test" input. (Note: the keys are playing the role of the A matrix in ridge-regression, each query is like the $\mathbf{x}$ we are testing on, and in place of the scalar y_i, we have an entire value vector for a multi-dimensional ridge-regression problem so the $\mathbf{y}$ vector is replaced with a matrix that has a row for each value vector.)

Assume that the cost of inverting a $d \times d$ matrix is $O(d^3)$ and the cost of multiplying two such matrices is also $O(d^3)$. Assume that a $d \times d$ matrix times either a d-dimensional row or column vector costs d^2 operations. You should assume $d < n$.

What is the computational cost of a non-causal ridge self-attention layer?

- ◯ $O(d^4)$
- ◯ $O(nd^2)$
- ◯ $O(n^2d^3)$
- ◯ $O(n^2d^2)$
- ◯ $O(n^2)$
- ◯ $O(1)$

(HINT: Do not forget that for a single d-dimensional query vector $\mathbf{q}$, attention needs to return a d-dimensional result.)

Solution: The answer is $O(nd^2)$.

The key to understanding this is that the computation of non-causal ridge self-attention can be broken up into two phases. In the first phase, we compute a $d \times d$ matrix W_* that represents the learned linear-transformation from all the n data points: $(\mathbf{k}_i, \mathbf{v}_i)$ in the attention layer. In the second phase, we simply compute $W_*^\top \mathbf{q}_i$ for each of the n queries. The second phase consists of n matrix-vector products and so clearly takes $O(nd^2)$ time. But what about the first phase?

For the first phase, we need to compute:

$$W_* = (\lambda I + A^\top A)^{-1} A^\top B \tag{2}$$

where $A = \begin{bmatrix} \mathbf{k}_1^\top \\ \mathbf{k}_2^\top \\ \vdots \\ \mathbf{k}_n^\top \end{bmatrix}$ and $B = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix}$ since we are computing (1) exactly d times for the d different entries in the value vectors, amortizing the load of computing all but the last part.

The first matrix $(\lambda I + A^\top A)$ can be computed in time $O(nd^2)$ and inverted in time $O(d^3)$. The second matrix $A^\top B = \sum_{i=1}^n \mathbf{k_i} \mathbf{v}_i^\top$ can be computed in time $O(nd^2)$. The matrix-matrix multiplication can also be performed in time $O(d^3)$. But since $n > d$ by assumption, each of the $O(d^3)$ times is also $O(nd^2)$ and this means that the total computation is just $O(nd^2)$ as well since there are a constant number of $O(nd^2)$ steps to perform.

In fact, you can make the computation of W_* effectively be $O(nd^2)$ steps when $n < d$ too. The matrix inversion can be computed starting with $(\lambda I)^{-1}$, then iterating with the Sherman–Morrison Formula (see a later part). Each step takes $O(d^2)$ time, and there are n steps, taking $O(nd^2)$ time in total.

 All Rights Reserved. This may not be publicly shared without explicit permission.

But all this is not required for this part of the problem.

(d) Assume that a ridge self-attention layer is used in a Transformer architecture and there is a downstream loss. **For which of these will backprop successfully pass gradients if we use ridge self-attention?**

- ☐ The ridge λ viewed as a learnable parameter for the self-attention layer.
- ☐ The keys
- ☐ The values
- ☐ The queries

Solution: Gradients pass through to all of these terms easily because all the steps involved are fully differentiable. As far as the queries are concerned, ridge self-attention is just a linear operation of multiplying by W_*. While the keys, values, and ridge λ do factor into the computation of W_* in a nonlinear way, it is a differentiable function.

(e) Now step back. There is a nice trick (called the Sherman–Morrison Formula) by which one can update the inverse of an invertible matrix to which you make a rank-1 update. Let M be an invertible square $d \times d$ matrix and let $\mathbf{u}, \mathbf{v}$ be two d-dimensional vectors. Then:

$$(M + \mathbf{u}\mathbf{v}^\top)^{-1} = M^{-1} - \frac{1}{1 + \mathbf{v}^\top M^{-1}\mathbf{u}}(M^{-1}\mathbf{u})(\mathbf{v}^\top M^{-1}) \tag{3}$$

Assume that a $d \times d$ matrix times either a d-dimensional row or column vector costs d^2 operations, and so does the evaluation of a dyad $\mathbf{u}\mathbf{v}^\top$. Assume that computing a Euclidean inner-product costs d operations. **Assuming that you already had M^{-1} in hand, what is the computational cost of one application of (3)?**

- ◯ $O(d^4)$
- ◯ $O(d^3)$
- ◯ $O(d^2)$
- ◯ $O(d)$
- ◯ $O(1)$

Solution: The answer is $O(d^2)$. This is because $M^{-1}\mathbf{u}$ can be computed in $O(d^2)$ time as a matrix-vector product. And so can $\mathbf{v}^\top M^{-1}$ by the same logic. Computing the dyad $(M^{-1}\mathbf{u})(\mathbf{v}^\top M^{-1})$ in (3) is therefore $O(d^2)$ in total. $\mathbf{v}^\top M^{-1}\mathbf{u}$ is just a Euclidean-inner product that we can compute in $O(d)$ time, and then dividing each entry of a vector by a scalar also costs $O(d)$ time. The total is thus $O(d^2)$ computation to do an inverse of a rank-1 update to a matrix.

(f) Consider implementing causal ridge-self-attention with a context length of n but where the pool of (key, value) vectors that one is querying at position t consists only of the t (key, value) pairs so far.

Describe explicitly how you would compute causal ridge-self-attention in a computationally efficient manner. Leverage your decomposition of the ridge-regression formula in part (b) of this problem together with the Sherman-Morrison formula from (3) to avoid having to do $O(n^2)$ computations while still calculating causal ridge-self-attention outputs correctly for all n positions in the context.

(HINT: Think recursively. What do you need to track from one time step to the next to avoid repeating work?)

Solution: Since the GPT-3 paper awakened the community to the power of in-context learning that such models seem to exhibit, understanding where this capability is coming from was one of the key pieces of the intellectual agenda. The fact that softmax-attention itself is close to soft-nearest-neighbor style online learning was one very important key clue. This overall problem should help you see that

 All Rights Reserved. This may not be publicly shared without explicit permission.

conceptually, one has alternatives. What this part does is show you why recurrences can also exhibit this kind of online learning capability.

Simple recurrences also intrinsically make the computation be $O(n)$ as far as the total context-length is concerned because you just make one forward pass through the data, doing the same amount of computation for every single position. By contrast, traditional softmax-attention has to do a linearly increasing amount of computation for every position since the pool of potential neighbor (key, value) pairs is growing and we need to compute the score of the position-specific query with respect to each of them. That is what makes traditional softmax-attention take $O(n^2)$ computation with context-length n. If we were to treat each position t independently and start from scratch, causal ridge-self-attention would also end up taking $O(n^2)$ computation. But we can do better by reusing work via recurrences.

The key here is to do things step by step. For causal ridge attention, we need to understand the $W_*(t)$ so that the output of attention is $W_*^\top(t)\mathbf{q}_t$. We need to be able to compute this. If we have the $d \times d$ matrix $W_*(t)$, then computing $W_*^\top(t)\mathbf{q}_t$ just costs $O(d^2)$ operations.

Adapting (2) for causal ridge-self-attention merely requires redefining the the relevant A and B matrices in that expression. Let $A_t = \begin{bmatrix} \mathbf{k}_1^\top \\ \mathbf{k}_2^\top \\ \vdots \\ \mathbf{k}_t^\top \end{bmatrix}$ and $B_t = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_t^\top \end{bmatrix}$ so that we just have the keys and values up to this point in time being used. Now, the counterpart to (2) is:

$$W_*(t) = \underbrace{(\lambda I + A_t^\top A_t)}_{M_t}{}^{-1} \underbrace{A_t^\top B_t}_{J_t} \tag{4}$$

There are two ways that one can proceed to get an algorithm. First (and more ambitiously), we can try to get an efficiently computable recursion for $W_*(t)$ itself:

$$\begin{aligned}
W_*(t) &= (M_t)^{-1} J_t && (5)\\
&= (\lambda I + \sum_{i=1}^{t} \mathbf{k}_i \mathbf{k}_i^\top)^{-1} (\sum_{i=1}^{t} \mathbf{k}_i \mathbf{v}_i^\top) && (6)\\
&= (M_{t-1} + \mathbf{k}_t \mathbf{k}_t^\top)^{-1} (J_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top) && (7)\\
&= (M_{t-1}^{-1} - \underbrace{(\frac{1}{1 + \mathbf{k}_t^\top M_{t-1}^{-1} \mathbf{k}_t} (M_{t-1}^{-1}\mathbf{k}_t))(\mathbf{k}_t^\top M_{t-1}^{-1})}_{U_t})(J_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top) && (8)\\
&= M_{t-1}^{-1} J_{t-1} - U_t J_{t-1} + (M_{t-1}^{-1}\mathbf{k}_t)\mathbf{v}_t^\top - (U_t \mathbf{k}_t)\mathbf{v}_t^\top && (9)
\end{aligned}$$

where we can consider $M_0 = \lambda I$ and hence $M_0^{-1} = \lambda^{-1} I$ and $J_0 = \mathbf{0}$. Notice that to compute (7) recursively, we just need to invert a rank-1 update to a matrix that we have already inverted, as well as do a rank-1 update to a matrix that we have already computed.

The expression (9) immediately reveals what we need to track recursively across time: M_t^{-1} and J_t for sure, and the product $W_*(t) = M_t^{-1} J_t$ as well since it can also be reused.

To be explicit: we have the following $O(d^2)$ computation-cost recursions for updating M_t^{-1} and J_t:

$$M_t^{-1} = M_{t-1}^{-1} - (\frac{1}{1 + \mathbf{k}_t^\top M_{t-1}^{-1} \mathbf{k}_t} (M_{t-1}^{-1}\mathbf{k}_t))(\mathbf{k}_t^\top M_{t-1}^{-1}), \tag{10}$$

$$J_t = J_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top. \tag{11}$$

 All Rights Reserved. This may not be publicly shared without explicit permission.

Going back to justify why the entire recursion for $W_*(t)$ given by (9) can also be done in $O(d^2)$ computation: Computing the first term $M_{t-1}^{-1}J_{t-1}$ is trivial since we already have it as $W_*(t-1)$. Doing the sums of all these terms in (9) together just costs us $O(d^2)$. Now, let us look at the individual other terms in (9).

Here, it is helpful to observe that U_t itself is actually a dyad inside:

$$U_t = \underbrace{(\frac{1}{1+\mathbf{k}_t^\top M_{t-1}^{-1}\mathbf{k}_t}(M_{t-1}^{-1}\mathbf{k}_t))}_{\mathbf{u}_a}\underbrace{(\mathbf{k}_t^\top M_{t-1}^{-1})}_{\mathbf{u}_b^\top} \tag{12}$$

and so the term $U_tJ_{t-1} = \mathbf{u}_a(\mathbf{u}_b^\top J_{t-1})$ that can be computed as a vector-matrix multiplication followed by a dyad computation. So this is $O(d^2)$ given computing $\mathbf{u}_a, \mathbf{u}_b^\top$ themselves, which we already know is $O(d^2)$ as well. So all this is $O(d^2)$.

By similar reasoning, the term $(U_t\mathbf{k}_t)\mathbf{v}_t^\top = \mathbf{u}_a((\mathbf{u}_b^\top\mathbf{k}_t)\mathbf{v}_t^\top)$ is also $O(d^2)$ as an inner-product ($O(d)$) followed by scaling a vector ($O(d)$) followed by a dyad calculation $O(d^2)$.

The final term to understand is $(M_{t-1}^{-1}\mathbf{k}_t)\mathbf{v}_t^\top$ which is clearly just a matrix-vector multiply followed by a dyad calculation — also $O(d^2)$.

This means that the entire operation of causal ridge-self-attention can be done in $O(nd^2)$ — the same order as non-causal ridge-self-attention.

The above answer is in great detail and gives us something that, strictly speaking, we do not really need: namely all the $W_*(t)$ for each time t as explicit $d \times d$ matrices. The other approach is to simply leverage the two $O(d^2)$ computation recursions (10) for updating M_t^{-1} — remembering to start at $M_0^{-1} = \lambda^{-1}I$ — and (11) for updating J_t — remembering to start at $J_0 = \mathbf{0}$. Then the causal ridge-self-attention answer for the query $\mathbf{q}_t$ can be computed as $W_*^\top(t)\mathbf{q}_t = J_t^\top((M_t^{-1})^\top\mathbf{q}_t)$ as a matrix-vector multiplication followed by another matrix-vector multiplication. This is $O(d^2)$ computation.

Either of these paths is sufficient for full credit on this problem, and we expected people to take the easier second path.

Notice what you have learned here: there are nonlinear[1] recursions with $O(d^2)$ dimensional state that permit one to exactly implement an attention-style mechanism that clearly does online learning from the given context. This is important since one of the ongoing and exciting efforts in the deep learning community is trying to figure out how to construct efficiently-computable recursions that can allow models to efficiently scale to very large[2] contexts.

(g) Many people consider important the ability to visualize the attention weights. For traditional softmax-attention, these are the outputs of the softmax for a given query vector. They tell you the exact amounts by which the attention outputs at this position are linear combinations of the values being fed in at this and other positions.

For ridge-attention and a given query vector, how would you compute the (possibly negative) weights associated to each of the value vectors in the context?

[1] The required nonlinearities in this case are analog multiplication and division. But it is known that we can approximate these with other nonlinearities combined with linear operations.

[2] We did not talk about it here in this course, but efficiently scaling to very large contexts is important for engineering settings where we want to use such generative models in conjunction with databases, etc. If we can count on the sophisticated generative model to figure out how to synthesize and distill relevant information from a lot of context, then we can use less sophisticated approaches to get approximately relevant stuff into the context. The larger the context we can productively use, the easier it is to make sure that the relevant information is there.

 All Rights Reserved. This may not be publicly shared without explicit permission.

Solution: For a given query, we have $W_*^\top \mathbf{q}$ as our output. Looking at (2), this is

$$W_*^\top \mathbf{q} = ((\lambda I + A^\top A)^{-1} A^\top B)^\top \mathbf{q} \tag{13}$$
$$= B^\top (A(\lambda I + A^\top A)^{-1}) \mathbf{q} \tag{14}$$

Since $B^\top = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \end{bmatrix}$, we can see that the column given by expression $(A(\lambda I + A^\top A)^{-1})\mathbf{q}$ gives us the counterpart of the attention weights. By itself, this is not very interpretable although it does answer the question. To make this slightly more interpretable, we can look at the i-th row of this vector and notice by the properties of matrix multiplication that it is $\mathbf{k}_i^\top (\lambda I + A^\top A)^{-1}\mathbf{q} = \langle \mathbf{q}, (\lambda I + A^\top A)^{-1}\mathbf{k}_i \rangle$. In other words, the i-th attention weight is the inner product between the query and a context-adjusted i-th key.

You could also do this calculation via kernel ridge regression which has the same predictions as regular ridge-regression, but uses a different formula which factors through an $n \times n$ matrix instead of $d \times d$. Recall that in place of $(A^\top A + \lambda I_{d\times d})^{-1} A^\top$, we use $A^\top (\lambda I_{n\times n} + AA^\top)^{-1}$ in kernel-ridge regression. Following the approach above, you would therefore get that the attention weights are: $(\lambda I_{n\times n} + AA^\top)^{-1} A\mathbf{q}$. Notice that $A\mathbf{q}$ is a column which consists of the same n inner-products between the n keys and the query as would be computed in standard soft-max attention. The difference is that in ridge-attention, instead of normalizing these inner-products into attention-weights by using a softmax, we have a "normalizing" matrix $(\lambda I_{n\times n} + AA^\top)^{-1}$ that only depends on the pairwise inner-products of all the keys with each other.

This problem hopefully serves to simultaneously de-mystify attention weights while also giving you leverage in thinking more clearly about what is going on.

2. 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)
```

 All Rights Reserved. This may not be publicly shared without explicit permission.

```
        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?)

Solution: a. d_model, b. d_model, c. d_model, d. Q, e. softmax, f. V, g. self.num_heads, h. W_o.

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

 All Rights Reserved. This may not be publicly shared without explicit permission.

```
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?)

Solution: Three changes are needed; changing only `W_o` is not enough.

- `W_o` must accept the wider concatenated heads: `self.W_o = nn.Linear(2 * d_model, d_model)`. Its output stays `d_model` so that the residual connection around the attention block still works.
- `split_heads` reshapes its input to `self.d_k = d_model // num_heads` entries per head, which fails on the `2 * d_model` values. The values need `2 * self.d_k` entries per head, e.g. `x.view(batch_size, seq_length, self.num_heads, -1)`. Queries and keys keep `self.d_k` entries per head, so the scaling by $\sqrt{d_k}$ does not change.
- `combine_heads` reshapes the concatenated heads to `self.d_model`; it must reshape to `2 * self.d_model` (or `-1`).

Any equivalent fix is correct, for example a separate per-head value size `self.d_v = 2 * self.d_k` used for the values.

3. 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]

```

 All Rights Reserved. This may not be publicly shared without explicit permission.

```
    # 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?**

◯ $[h, d, q]$ ◯ $[d, q]$ ◯ $[h, q]$

Solution: ■ $[h, d, q]$ ◯ $[d, q]$ ◯ $[h, q]$

 All Rights Reserved. This may not be publicly shared without explicit permission.

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

◯ $[h, d, k]$ ◯ $[d, k]$ ◯ $[h, k]$

Solution: ◯ $[h, d, k]$ ■ $[d, k]$ ◯ $[h, k]$

(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)`

Solution: ◯ `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)`

Solution: ■ `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)`

Solution: ◯ `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)?**

Solution: Compute: $O(bd^2 + bnd)$

- $O(bd^2)$ from projections (query, key, value, output)
- $O(bnd)$ from attention (dot products with n cached keys/values)

Note that although the same order of magnitude, the projections are a bit cheaper in MQA because we only have to compute one set of keys and values — not h of them. We do have to compute h different queries however, so in order notation, this is just a constant factor difference.

 All Rights Reserved. This may not be publicly shared without explicit permission.

(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)?**

Solution: Memory: $O(d^2 + bd + bnk)$

- $O(d^2)$ to read weights
- $O(bd)$ to read/write current token embeddings (x, q, k, v, o, y)
- $O(bnk)$ to read cached keys/values (reduced from $O(bnd)$ in MHA due to shared K/V)

Because we only need to read one set of weights to form keys and values (instead of separate such weights for each head), there is a savings in the first term that is being hidden by the big O notation. Similarly, there are fewer keys and values to write, but the $O(bd)$ term is dominated by writing the outputs.

4. 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.

Solution: Notice that GPT is paying attention to previous words in the sequence. Specifically, in the first example, the word 'ran' attends to the token 'dog' because it is important for the model to know who or what is doing the running. In the second example, notice that in the default view (layer 0, head 0) GPT is paying more attention to the subject of the sentence 'dog', rather than the object of the prepositional phrase 'car' (other heads differ, so answers may vary). In the example where GPT is tasked with keeping track of past information, the model is indeed appropriately making a connection between the name mentioned previously and the vague pronoun reference in the second sentence. The queries, keys, and values further support this fact that GPT is successful in identifying linguistic patterns.

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?**

Solution: Students may notice any number of patterns. Here's an example. As you look through the layers of the network, you may notice that GPT tends to place a lot of attention on the first word in the sentence. This is because when the model doesn't know what linguistic patterns are interesting or relevant to the current setting, it looks to the start of the sentence for answers. If you are interested in seeing some more interesting patterns, feel free to play around with this tool some more, or read this interesting blog post.

(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?**

Solution: At some layers, words attend to the immediate next or previous token. At others, there are a lot of attentions to the CLS token. Another occurence is that that words will attend to other instances of the word in the sentence. In the middle and later layers (roughly layers 3–11), many heads put most of their attention on the [SEP] tokens.

 All Rights Reserved. This may not be publicly shared without explicit permission.

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

Solution: BERT is a bidirectional model, whereas GPT only attends to tokens one at a time. When looking at the words that the attention heads are paying atteniont to this difference becomes obvious.

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?**

Solution: The word "play" in this context has significantly different meanings based on the context. While the overall attention patterns look very similar, in some layers you can see that keys corresponding to “play” look different between the two examples.

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?**

Solution: BERT uses bidirectionality in order to attend backwards. For example, SEP tokens look for CLS tokens even though the attention head at layer 2, head 0 mostly consists of words that look at the next work. And yes! BERT is very powerful because the training is self-supervised, and so we do not need to manually label data–pretraining BERT gives it access to larges amounts of data that it can generalize from to form useful representations. If you are interested in seeing some more interesting patterns, feel free to play around with this tool some more, or read this interesting blog post.

(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?**

Solution: Students should notice different patterns being learned by the different attention heads in the rows of the visualization. Some interesting patterns include the following: an attention head focuses on the special SEP and CLS tokens in the text; an attention head focuses on all words in the text equally; an attention head focuses on the word that was directly before it; etc. The attention heads are able to focus on different patterns as their queries, keys, and values are calculated independently and then combined together to produce a final attention score that is eventually used downstream.

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

Solution: Students should be able to see the different patterns in the visualization. Some are listed in the solution for the part above.

(d) Visualizing untrained attention weights

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

Solution: All of the words pay equal attention to all of the other words. The untrained BERT has no way of distinguishing which words might be more important, and which words are less so.

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?**

Solution: As seen from the previous parts of the notebook, BERT pays particular attention to word before, the word after, separators, and words that have similar meanings. We expect gradients at these steps to be higher, since these are the features that the model eventually learns.

 All Rights Reserved. This may not be publicly shared without explicit permission.

(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.

Solution: We understand that making interpretable observations from large, parametrized models can be challenging, and hope you appreciate this fact too.

5. 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 [3].

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 [4]. In particular, attention can be seen as applying a kernel over the inputs 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[5] is a function $T : \mathbb{R}^{N\times F} \to \mathbb{R}^{N\times F}$ defined by the composition of L transformer layers $T_1(\cdot), \cdots, T_L(\cdot)$ as follows,

$$T_l(x) = f_l(A_l(x) + x). \tag{15}$$

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,

$$\begin{aligned} Q = xW_Q, K = xW_K, V = xW_V, \\ A_l(x) = V' = \text{softmax}(\frac{QK^T}{\sqrt{D}})V. \end{aligned} \tag{16}$$

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 16 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 $i\cdot$ returns the i-th row as a vector and subscripting with $\cdot j$ returns the j-th column, we can write a generalized

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

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

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

 All Rights Reserved. This may not be publicly shared without explicit permission.

attention equation for any similarity function as follows,

$$V'_{i\cdot} = \frac{\sum_{j=1}^{N} \text{sim}\left(Q_{i\cdot}, K_{j\cdot}\right) V_{j\cdot}}{\sum_{j=1}^{N} \text{sim}\left(Q_{i\cdot}, K_{j\cdot}\right)}. \tag{17}$$

Equation 17 is equivalent to equation 16 if we substitute the similarity function with $\text{sim}_{\text{softmax}}(q,k) = \exp(\frac{q^T k}{\sqrt{D}})$. This can lead to

$$V'_{i\cdot} = \frac{\sum_{j=1}^{N} \exp(\frac{Q_{i\cdot} K_{j\cdot}^T}{\sqrt{D}}) V_{j\cdot}}{\sum_{j=1}^{N} \exp(\frac{Q_{i\cdot} K_{j\cdot}^T}{\sqrt{D}})}. \tag{18}$$

For computing the resulting self-attended feature $A_l(x) = V'$, we need to compute all $V'_{i\cdot}$ $i \in 1, ..., N$ in equation 18.

(a) **Identify the conditions that needs to be met by the** sim **function to ensure that** $V_{i\cdot}$ **in Equation 17 remains finite (the denominator never reaches zero).**

Solution: In order for $V_{i\cdot}$ in Equation 17 to remain finite, it is necessary that $\sum_{j=1}^{N} \text{sim}\left(Q_{i\cdot}, K_{j\cdot}\right) \neq 0$ for all possible values of Q and K. This condition implies that the function value of sim must always have the same sign (positive or negative).

For the purposes of this problem, we will only consider the case where sim consists of *kernels* $k(x,y) : \mathbb{R}^F \times \mathbb{R}^F \to \mathbb{R}^+$ that yield positive values.

(b) The definition of attention in equation 17 is generic and can be used to define several other attention implementations.

(i) One potential attention variant is the "polynomial kernel attention", where the similarity function as $\text{sim}(q,k)$ is measured by polynomial kernel $\mathcal{K}$ [6]. **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.)** .

Solution: Quadratic kernel attention is $\text{sim}(q,k) = \left(q^T k + 1\right)^2 = \phi(q)^T \phi(k)$.

(ii) One benefit of using kernelized attention is that we can represent a kernel using a feature map $\phi(\cdot)$ [7]. **Derive the corresponding feature map** $\phi(\cdot)$ **for the quadratic kernel.**

Solution: The feature map $\phi(\cdot)$ can be expressed as

$$\phi(x) = [1, \sqrt{2}x_1, \sqrt{2}x_2, ..., \sqrt{2}x_D, x_1^2, x_2^2, ..., x_D^2, \sqrt{2}x_1x_2, ..., \sqrt{2}x_{D-1}x_D]^T$$

(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 17 with feature map** $\phi(\cdot)$**.**

Solution: The general kernel attention can be rewritten as

$$V'_{i\cdot} = \frac{\sum_{j=1}^{N} \phi(Q_{i\cdot}) \phi\left(K_{j\cdot}\right)^T V_{j\cdot}}{\sum_{j=1}^{N} \phi(Q_{i\cdot}) \phi\left(K_{j\cdot}\right)^T},$$

[6] https://en.wikipedia.org/wiki/Polynomial_kernel
[7] https://en.wikipedia.org/wiki/Kernel_method

 All Rights Reserved. This may not be publicly shared without explicit permission.

(c) We can rewrite the softmax attention in terms of equation 17 as equation 18. **For all the $V'_{i\cdot}$ ($i \in \{1, ..., N\}$), 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

Solution: The computational graph can be illustrated with the following pseudocode:

```
for i in range(N):
    for j in range(N):
        S[i, j] = Q[i, :] @ K[j, :].T / sqrt(D)
for i in range(N):
    Z = 0
    for j in range(N):
        Z += exp(S[i, j])
    for j in range(N):
        A[i, j] = exp(S[i, j]) / Z
for i in range(N):
    O[i, :] = 0
    for j in range(N):
        O[i, :] += A[i, j] * V[j, :]
```

Time: Therefore, the computational cost of each double-layer for-loop is $\mathcal{O}(N^2D)$, $\mathcal{O}(N^2D)$, and $\mathcal{O}(N^2M)$, respectively, So the total time complexity is $\mathcal{O}(N^2(D+M))$ (or equivalently, $\mathcal{O}(N^2 \max(D, M))$).
Space: `Q`, `K` takes $\mathcal{O}(ND)$ memory, `S`, `A` needs $\mathcal{O}(N^2)$ memory, and `V`, `O` needs $\mathcal{O}(NM)$ memory. So the total space complexity is $\mathcal{O}(N(N + M + D))$ (or equivalently, $\mathcal{O}(N \max(N, M, D))$).

(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 17 with $\text{sim}(x, y) = \mathcal{K}(x, y) = (\phi(Q_{i\cdot})^T \phi(K_{j\cdot}))$ in part (b). We can then further simplify it by making use of the associative property of matrix multiplication to

$$V'_{i\cdot} = \frac{\phi(Q_{i\cdot}) \sum_{j=1}^{N} \phi(K_{j\cdot})^T V_{j\cdot}}{\phi(Q_{i\cdot}) \sum_{j=1}^{N} \phi(K_{j\cdot})^T}. \tag{19}$$

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 (19).

Solution: The computational graph can be illustrated with the following pseudocode:

```
U[:, :] = 0  # shape: [C, M]
for j in range(N):
    U[:, :] += phi(K[j, :]).T @ V[j, :]
Z[:] = 0  # shape: [C]
for j in range(N):
    Z[:] += phi(K[j, :]).T
for i in range(N):
    O[i, :] = phi(Q[i, :]) @ U[:, :]
    O[i, :] /= phi(Q[i, :]) @ Z[:]
```

 All Rights Reserved. This may not be publicly shared without explicit permission.

Time: the computational cost of each step is $\mathcal{O}(NCM)$, $\mathcal{O}(NC)$, $\mathcal{O}(NCM)$, respectively. So the total computational cost is $\mathcal{O}(NCM)$. For the quadratic kernel, we have $C = \mathcal{O}\left(D^2\right)$, and apply $M \approx D$, then the time complexity is $\mathcal{O}\left(ND^3\right)$. If $N >> D^2$, kernelized linear attention with a quadratic polynomial kernel is faster than softmax attention.

Space: `Q`, `K` takes $\mathcal{O}(ND)$ memory, `U` needs $\mathcal{O}(CM)$ memory, and `V`, `O` needs $\mathcal{O}(NM)$ memory. So the total space complexity is $\mathcal{O}\left(N(D+M)+CM\right)$. For the quadratic kernel, we have $C = \mathcal{O}\left(D^2\right)$, and apply $M \approx D$, then the space complexity is $\mathcal{O}\left(ND + D^3\right)$ (or equivalently $\mathcal{O}\left(D\max(N, D^2)\right)$). If $N >> D^2$, kernelized linear attention uses much less memory than softmax attention.

6. Coding Question: Summarization

Please follow the instructions in this notebook. You will implement a Transformer using fundamental building blocks in PyTorch. You'll apply the Transformer encoder-decoder model to a sequence-to-sequence NLP task: document summarization. Refer to the Attention is All You Need paper for details on the model architecture.

Once you finished with the notebook, answer the following questions in your submission of the written assignment:

(a) **Please submit the screenshots of the training loss and the validation loss displayed on Tensorboard.**

Solution: They should look like:

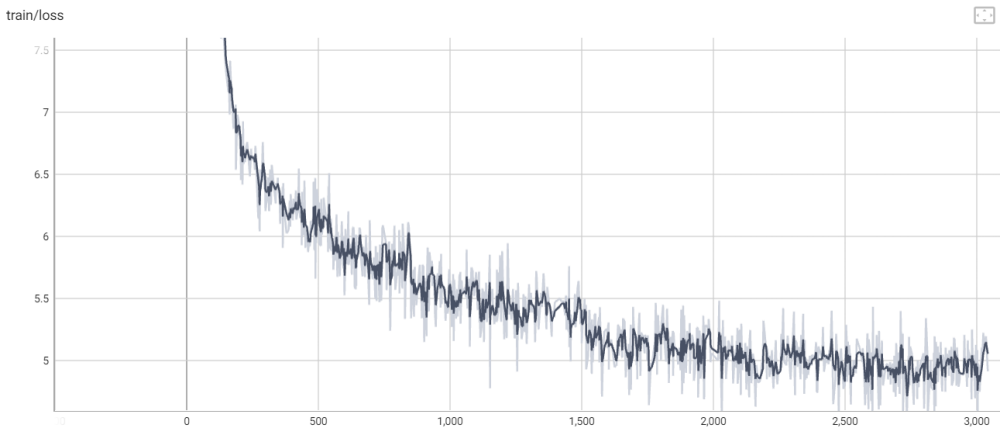


Figure 1: Training loss.

 All Rights Reserved. This may not be publicly shared without explicit permission.

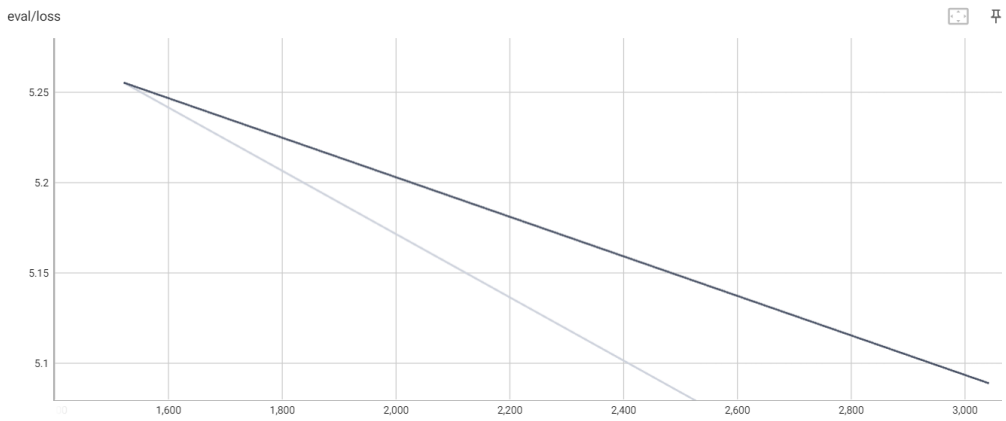


Figure 2: Validation loss.

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: Matteo Guarrera, Mert Cemri, Sizhe Chen, Gireeja Ranade, Anant Sahai, Yossi Gandelsman, Naman Jain, Dhruv Shah, Shivam Singhal, Kevin Li, Olivia Watkins, Bryan Wu, Sheng Shen, Shaojie Bai, Angelos Katharopoulos, Hao Peng, Linyuan Gong, Hao Liu, Allie Gu, CS182 Staff from past semesters.

 All Rights Reserved. This may not be publicly shared without explicit permission.