EECS 182 Deep Neural Networks
Fall 2026 Gireeja Ranade

Homework 9

1. Coding SSM Forward

The SSM forward pass can be implemented using recurrent and convolution approaches. In this problem, we will implement the two approaches and compare their efficiency. Particularly, you are given the following update equation:

$$\mathbf{h_{t+1}} = (W \cdot \mathbf{h_t} + U \cdot \mathbf{x}_t + \mathbf{b})$$

You are given all the input $\mathbf{x}_t$ for $t = 0, 1, \ldots, T-1$ and the initial hidden state $\mathbf{h}_0 = 0$. You need to implement the forward pass for the SSM computing all the hidden states $\mathbf{h}_t$ for $t = 1, 2, \ldots, T$. Use batch size N, input dimension D, and hidden dimension H.

We will complete the notebooks q_coding_ssm_forward_cpu.ipynb and q_coding_ssm_forward_gpu.ipynb on the CPU and GPU respectively. Setup instructions and starter code are in the HW09 code folder. Use the small default timing grids. For the GPU part, use CUDA or Apple Silicon MPS; if unavailable, use the notebook's CPU correctness fallback and state that no GPU timing was measured. Compare matched dimensions, and use the supplied warmup and device synchronization when timing.

CPU Implementation:

(a) Implement the forward pass by unrolling the SSM in time in a recurrent manner.

Solution: Initialize an $N \times H$ zero array h. For $t = 0, \ldots, T-1$, set $h \leftarrow hW^\top + x[:, t, :]U^\top + b$ and save h. Stack the saved arrays along the time axis to return shape (N, T, H).

(b) Implement the forward pass by using a convolution-based approach. You will create the kernel K and use `nn.Conv1d` (or functional `conv1d`) to apply the kernel to the projected input $s_t = Ux_t + b$. Hint: You can create the kernel using a divide-and-conquer method. Assume T is a positive power of two, include the bias at every step, and use causal padding.

Solution: Unrolling gives $h_{t+1} = \sum_{j=0}^{t} W^{t-j} s_j$. Build $I, W, \ldots, W^{T-1}$ by doubling: append W^m times the current block $I, \ldots, W^{m-1}$, then square W^m for the next block. Since `conv1d` uses cross-correlation, set $K[:, :, j] = W^{T-1-j}$. Transpose $s = xU^\top + b$ to (N, H, T), pad $T-1$ zeros on the left, apply `conv1d`, and transpose the result back to (N, T, H).

(c) Compare the runtime of the two approaches and explain your findings.

Solution: For a batch of N sequences, input projection costs $O(NTDH)$ in both methods. Dense recurrence adds $O(NTH^2)$ work. Constructing the shared dense powers costs $O(TH^3)$ classical arithmetic and stores $O(TH^2)$ entries; direct length-T convolution adds $O(NH^2T^2)$ work. CPU vectorization and multiple threads can exploit parallelism. Which method is faster depends on dimensions, implementation, and hardware; explain the measured result.

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

GPU Implementation:

(d) Now implement the forward pass on the GPU. You can copy your existing implementation to the GPU notebook.

Solution: Use the same recurrence and convolution implementations from parts (a) and (b), with W, U, b, x and all intermediate tensors on the same CUDA or MPS device. Use the notebook's common float32 precision policy.

(e) Compare the runtime of the two approaches and explain your findings.

Solution: GPUs can parallelize convolution over time, while recurrence has a sequential time dependency. This can favor convolution in some regimes, but dense kernel construction, memory use, and direct-convolution work grow with H and T. Launch overhead also matters for small cases. There is no universal winner; report synchronized timings including kernel construction and explain the observed tradeoff.

(f) To optimize the convolution-based approach, we will constrain W to be diagonal and leverage depthwise convolutions. Implement the new forward pass under this constraint and now compare the runtimes against a diagonal recurrence using elementwise multiplication.

Solution: Let $r = \mathrm{diag}(W)$. Replace $hW^\top$ by the elementwise product $h \odot r$ in the recurrence. For convolution, use $K[i, 0, j] = r_i^{T-1-j}$, the same projected input and causal padding as in part (b), and `groups=H`. Input projection costs $O(NTDH)$; diagonal recurrence adds $O(NTH)$ work. The shared depthwise kernel takes $O(TH)$ construction work and storage, and direct depthwise convolution adds $O(NHT^2)$ work. Either method may be faster depending on dimensions and backend; measurements for one grid do not establish a speedup for every H.

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

2. 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_i and standard deviation σ, and the same is true for k.

Attention applies softmax to query–key dot products. When their variance grows with dimension, softmax can become more concentrated and its gradients smaller. This problem motivates scaling the dot products to control their variance.

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

Solution: $\mathbb{E}[q^T k] = \mathbb{E}[\sum_{i=1}^d q_i k_i] = \sum_{i=1}^d \mathbb{E}[q_i k_i] = \sum_{i=1}^d \mathbb{E}[q_i]\mathbb{E}[k_i] = \sum_{i=1}^d \mu_i^2 = ||\mu||_2^2$

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

Solution: Here $\Sigma_q = \text{Cov}(q) = \sigma^2 I$ and $\Sigma_k = \text{Cov}(k) = \sigma^2 I$. $\text{Var}(q^T k) = \mathbb{E}[(q^T k)^2] - \mathbb{E}[q^T k]^2 = \mu^T \Sigma_q \mu + \mu^T \Sigma_k \mu + \text{tr}(\Sigma_q \Sigma_k) = 2\sigma^2 ||\mu||_2^2 + d\sigma^4 = d$ for $\mu = 0$ and $\sigma = 1$.
(here's an alternate way of solving this)

$$
\begin{aligned}
\text{Var}(q^T k) &= \text{Var}(\sum_{i=1}^d (q_i k_i)) && \text{// def of dot product} \\
&= \sum_{i=1}^d (\text{Var}(q_i k_i)) && \text{// all } q_i \text{ and } k_i \text{ are independent} \\
&= d\text{Var}(q_1 k_1) && \text{// same variance for all i} \\
&= d(\mathbb{E}[q_1]^2 \text{Var}(k_1) + \mathbb{E}[k_1]^2 \text{Var}(q_1) + \text{Var}(q_1)\text{Var}(k_1)) && \text{// independent product} \\
&= d * \text{Var}(q_1)\text{Var}(k_1) && \text{// all the expectations are 0} \\
&= d\sigma^4 \\
&= d
\end{aligned}
$$

(c) Continue to use the setting in part (b), where $\mu = 0$ and $\sigma = 1$. Let $s > 0$ be a deterministic 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 1. What should s be in terms of d?

Solution: From part (a), we know that $\mathbb{E}[q^T k] = \sum_{i=1}^d \mu_i^2 = 0$. From part (b), we know that $\text{Var}[q^T k] = d$. So for $\mathbb{E}[\frac{q^T k}{s}]$ to be 0, and $\text{Var}(\frac{q^T k}{s}) = \frac{d}{s^2}$ to be 1, we can have $s = \sqrt{d}$

3. 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. In the notation below, argmax denotes the one-hot encoding of the maximizing index.

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

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

Solution: Compute inner products of query with the keys:

$$\langle [1,1,2]^\top, [1,2,0]^\top \rangle = 3 \qquad \langle [1,1,2]^\top, [0,3,4]^\top \rangle = 11$$
$$\langle [1,1,2]^\top, [5,0,0]^\top \rangle = 5 \qquad \langle [1,1,2]^\top, [0,0,1]^\top \rangle = 2$$

We take argmax rather than softmax, and have $\text{argmax}([3, 11, 5, 2]) = [0, 1, 0, 0]$

Now, take weighted sum of value vectors (in this case all are zeroed out except for the one corresponding to the highest dot-product between query and key)

$0 \cdot [2, 0, 1]^\top + 1 \cdot [1, 4, 3]^\top + 0 \cdot [0, -1, 4]^\top + 0 \cdot [1, 0, -1]^\top$

So final output is $\begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$

(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?)

Solution: With ordinary backpropagation, hard selection is locally constant in the scores wherever the maximizer is unique. Thus gradients through the selection operation to the queries and keys are zero almost everywhere; the ordinary derivative is undefined at ties where the winner changes. This path supplies no useful signal to improve the query/key projections. Gradients still flow to the selected value and its producing parameters. Hard attention can be trained with a differentiable relaxation or another gradient estimator, but argmax alone does not provide that mechanism.

Contributors: Matteo Guarrera, Mert Cemri, Sizhe Chen, Gireeja Ranade, Anant Sahai, Naman Jain, Qiyang Li, Sheng Shen, David M. Chan, Saagar Sanghavi.

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