Labs
Machine Learning Course
Fall 2025

EPFL
School of Computer and Communication Sciences
Bob West
epfml.github.io/cs433-2025/

Problem Set 8, Nov 6, 2025 (Solutions to Theory Questions)

Problem 1 (Computation and Memory Requirements of Neural Networks):

Let's consider a fully connected neural network with L layers in total, all of width K. The input is a mini-batch of size $n \times K$. For this exercise we will only consider the matrix multiplications, ignoring biases and activation functions.

- How many multiplications are needed in a forward pass (inference)?
- How many multiplications are needed in a forward + backward pass (training)?
- How much memory is needed for an inference forward pass? The memory needed is the sum of the memory needed for activations and weights. Assume activations are deleted as soon as they are no longer required.
- How much memory is needed for a training forward + backward pass? Note that during training we additionally use the forward activations for the computation of the derivatives.

Solution:

We have a sequence of L matrix multiplications $Y = XW$ where $X \in \mathbb{R}^{n\times K}$ and $W \in \mathbb{R}^{K\times K}$. In the backward pass we compute derivatives w.r.t. the input $\delta_X = \delta_Y W^\top$ and the weights $\delta_W = X^\top \delta_Y$ where δ_Y are the derivatives w.r.t. the output Y. The derivatives have the same shapes as the original variables. The number of multiplications for each of the tree potential matrix multiplications is nK^2. The number of parameters in W is K^2.

During inference we compute L matrix multiplications for a total of nK^2L multiplications. We need to store all parameter matrices and X and Y for a single layer at a time. The memory requirements are therefore $LK^2 + 2nK$ values.

During training we additionally compute the matrix multiplications in the backward pass for a total of $3nK^2L$ multiplications. We store the same parameters but now we also need to store their gradients. Since we need X for the computation of δ_W we also need to store this for each layer. In total our memory requirements are roughly $2LK^2 + LnK$ values.

Note that if n is large, training can require significantly more memory than inference. The optimizer might also keep more values for each weight than the gradient, such as the first and second moments of the gradients. This further increases the memory requirements. The computation cost of training is roughly $3\times$ higher than inference per batch.

Problem 2 (Variance Preserving Weight Initialization for ReLUs):

When training neural networks it is desirable to keep the variance of activations roughly constant across layers. Let's assume we have $y = \mathbf{x}^\top \mathbf{w}$ where $\mathbf{x} = \text{ReLU}(\mathbf{z})$, $\mathbf{z} \in \mathbb{R}^d$ and $\mathbf{w} \in \mathbb{R}^d$. Further assume that all elements $\{w_i\}_{i=1}^d$ and $\{z_i\}_{i=1}^d$ are independent with $w_i \sim \mathcal{N}(0, \sigma^2)$ and $z_i \sim \mathcal{N}(0, 1)$.

Derive $\text{Var}[y]$ as a function of d and σ i.e. how should we set σ to have $\text{Var}[y] = 1$.

Hint: Remember the law of total expecation i.e. $\text{E}_A[A] = \text{E}_B[\text{E}_A[A|B]]$

Solution:

$$\begin{aligned}
\mathsf{Var}[y] &= \mathsf{Var}\left[\sum_{i=1}^{d} w_i x_i\right] \\
&= \sum_{i=1}^{d} \mathsf{Var}\left[w_i x_i\right] && \text{independent terms} \\
&= \sum_{i=1}^{d} \mathsf{E}[(w_i x_i)^2] - \mathsf{E}[w_i x_i]^2 && \text{formula for variance} \\
&= \sum_{i=1}^{d} \mathsf{E}[w_i^2]\mathsf{E}[x_i^2] - \mathsf{E}[w_i]^2\mathsf{E}[x_i]^2 && \text{independence} \\
&= \sum_{i=1}^{d} \mathsf{E}[w_i^2]\mathsf{E}[x_i^2] && \text{since } \mathsf{E}[w_i] = 0 \\
&= d\sigma^2 \mathsf{E}[x_i^2] \\
&= d\sigma^2 \mathsf{E}[\mathsf{ReLU}(z_i)^2 | z_i \geq 0] p(z_i \geq 0) + \mathsf{E}[\mathsf{ReLU}(z_i)^2 | z_i < 0] p(z_i < 0) && \text{law of total expectation} \\
&= \frac{1}{2} d\sigma^2 \mathsf{E}[z_i^2] && \text{ReLU is identity or zero} \\
&= \frac{1}{2} d\sigma^2
\end{aligned}$$

We should thus set $\sigma = \sqrt{2/d}$ to preserve variance.

Problem 3 (Adam and SGDM):

SGD with momentum (SGDM) and Adam are two very commonly used optimizers in deep learning. Both are example of first order optimization methods that update the weights based on their gradients after some processing. The two algorithms are given below. Note that both of these algorithms act on each scalar parameter independently, and do not consider whether a parameter is a part of a larger vector/matrix/tensor.

Adam:

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1 - \beta_1) \nabla_w L^{(t)} \tag{1}$$

$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1 - \beta_2)(\nabla_w L^{(t)})^2 \tag{2}$$

$$\hat{m}_w = \frac{m_w^{(t+1)}}{1 - \beta_1^{t+1}} \tag{3}$$

$$\hat{v}_w = \frac{v_w^{(t+1)}}{1 - \beta_2^{t+1}} \tag{4}$$

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon} \tag{5}$$

SGDM:

$$m_w^{(t+1)} \leftarrow \beta m_w^{(t)} + \nabla_w L^{(t)} \tag{6}$$

$$w^{(t+1)} \leftarrow w^{(t)} - \eta m_w^{(t+1)} \tag{7}$$

For both algorithms, $L^{(t)}$ is the loss for time t (typically this is the loss for a mini-batch of samples), and $w^{(t)}$ represents the value of the parameter at step t. The algorithm shows an update for a single parameter but all model parameters are updated in the same way for each timestep t. Both optimizers use an exponential moving average of the gradient called momentum (represented by $m_w^{(t)}$). Adam additionally uses an exponential moving average of the square gradient ($v_w^{(t)}$) and also computes a "bias correction" for m and v given by $\hat{m}$ and $\hat{v}$. In both cases we consider the intial state to be zero, i.e. $m_w^{(0)} = v_w^{(0)} = 0$. The hyperparameters and their possible values are $\eta \geq 0, 0 < \beta_1 < 1, 0 < \beta_2 < 1, \epsilon \geq 0$ for Adam and $\eta \geq 0, 0 < \beta < 1$ for SGDM.

1. How many values does each optimizer need to store for a given parameter to perform the next update? This factor determines the memory usage of the optimizer.

2. Let's assume the gradient is a constant $\nabla_w L^{(t)} = c > 0$ for all $t \geq 0$ and $\epsilon = 0$. Compute the value of $w, m_w, v_w, \hat{m}_w, \hat{v}_w$ for timestep t and both optimizers (where applicable). Assume $w^{(0)} = 0$ for this question. How does w depend on c in each case?

Solution:

Part 1: Adam needs to store m_w, v_w and of course the weights w if we consider them a part of the optimizer and not the model. Aside from the weights, we only need m_w for SGDM. The extra memory (ignoring the weights) required for Adam is twice that of SGDM.

Part 2: For SGDM we have:
$m_w^{(t)} = c + \beta m_w^{(t-1)} = \ldots = c + \beta c + \beta^2 c + \ldots + \beta^{t-1} c = c \sum_{k=0}^{t-1} \beta^k = c \frac{1-\beta^t}{1-\beta}$

$w^{(t)} = -\eta \sum_{k=0}^{t} m_w^{(k)} = -\frac{\eta c}{1-\beta} \sum_{k=0}^{t} (1-\beta^k) = -\frac{\eta c}{1-\beta} \left(t + 1 - \frac{1-\beta^{t+1}}{1-\beta} \right)$

For Adam we have:
$m_w^{(t)} = (1-\beta_1)c + \beta_1 m_w^{(t-1)} = \ldots = (1-\beta_1) \cdot c \cdot (1 + \beta_1 + \beta_1^2 + \ldots + \beta_1^{t-1}) = (1-\beta_1) c \sum_{k=0}^{t-1} \beta_1^k = c \cdot (1 - \beta_1^t)$
With the bias correction this gives: $\hat{m}_w = c$.
Similarly we get $v_w^{(t)} = c^2 \cdot (1 - \beta_2^t)$ and $\hat{v} = c^2$.
Then $w^{(t)} = \sum_{k=1}^{t} \left(-\eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w + \epsilon}} \right) = -\eta t$ (since $\epsilon = 0$).
Note that this does not depend on c at all. This highlights an important property of Adam, it is invariant to a constant scaling of the gradient. Thus it is for example invariant to scaling of the loss function (all of this assuming ϵ is sufficiently small).

Problem 4 (Receptive Field of Convolutions):

Convolutions can occur in one or more dimensions. In class you learned about 2D convolutions but both 1D and 3D convolutions are used in certain areas as well (for signals of a corresponding dimension). 1D convolutions are easy to visualize and many insights about them generalize to higher dimensions. You can view a 1D convolution as a special case of 2D convolution where the height of the input and filter is equal to 1.

In this exercise we will explore how the size of the output depends on the input size and parameters of a convolution in 1D. We will then use this to analyze the receptive field of a convolutional networks. The receptive field of a given activation is the area of the input that can affect its value. This is important to keep in mind when working with convolutional networks since the receptive field must be sufficiently large for certain features to be learned. For example, if your convolutional network was looking for a certain phrase in an audio signal, the receptive field of the later neurons should be sufficiently long to cover the length of the phrase.

The output size of a 1D convolution is depends on the dimensions of the input as well as the kernel size K, the padding P and the stride S. Padding is applied to both sides of the input signal and adds P values to each side, typically zeros (but various other forms of padding also exist). After adding a given amount of padding, a convolution only computes outputs where the filter can be fully "overlayed" on the padded input signal. A convolution with stride S only computes every S-th element of the output (starting with the first valid position on the edge). In modern networks we often use strided convolutions instead of adding pooling layers.

- Let's assume we have a 1D convolution with input X of width W_{in}, a kernel size of K, padding P and stride S. What is the size W_{in} of the output Y?
- Given an output size W_{out} for the convolution above, what is the minimum size of the input, W_{in}?
- Given a sequence of L convolutions with kernel sizes $K^{(1)}, \ldots K^{(L)}$, padding $P^{(1)}, \ldots P^{(L)}$, and strides $S^{(1)}, \ldots S^{(L)}$, what is the receptive field of an output element of the last convolution? You can assume that the input is larger than the receptive field (otherwise the definition is unclear).

  Hint: Does padding affect the receptive field? Start with an output width of 1 and work your way backwards using the results of the previous parts. You don't have to simplify the resulting recurrence relation.

Solution:

Part 1: Let's first consider a convolution with no stride or padding. In this case there are $W_{in} - K + 1$ positions where the filter can be overlayed on top of the input. Each such position gives one output element. When we add padding we effectively increase the width of the input by $2P$ and get $W_{in} + 2P - K + 1$ output elements in total. With a stride S, we only compute every S-th output, starting with the first one at the edge of the input (and not one in the middle for example). In total we get the first output and then overy S-th one of the remaining

$W_{in} + 2P - K$ outputs for a total of $W_{out} = (W_{in} + 2P - K)/S + 1$ outputs. This assumes that S divides the output exactly, otherwise it should be rounded down (integer division), but generally we want the division to be perfect for symmetry.

Part 2: Previously we had $W_{out} = (W_{in}+2P-K)/S+1$. Solving this for W_{in} gives $W_{in} = S\cdot(W_{out}-1)+K-2P$. This is the minimum input size which corresponds to the case where the division by S was perfect.

Part 3: In the previous part we found an equation to get the minimum input size for a given output size and configuration for the convolution. To find the receptive field of the last layer we should start with an output of width one and then recursively use the previous formula to find what the first input width should be. However, we must compute the input size assuming no padding. This is because padding only extends the input size, it does not affect how large the area that affects each output is. To see this we can use the previous formula for $W_{out} = 1, K = 3, P = 1$ then $W_{in} = 1$ but each output is clearly a function of $K = 3$ elements of the extended input.

Since the input width of layer $l + 1$ is equal to the output width of layer l we can write this as:

$$W^{(l)} = S^{(l+1)} \cdot (W^{(l+1)} - 1) + K^{(l+1)}, \qquad W^{(L)} = 1 \tag{8}$$

where we want to find $W^{(0)}$.

Expanding the recurrence we get:

$$\begin{aligned}
W^{(l)} &= S^{(l+1)}W^{l+1} + (K^{(l+1)} - S^{(l+1)}) && (9)\\
&= S^{(l+1)}\left(S^{(l+2)}W^{l+2} + (K^{(l+2)} - S^{(l+2)})\right) + (K^{(l+1)} - S^{(l+1)}) && (10)\\
&= S^{(l+1)}\left(S^{(l+2)}\left(S^{(l+3)}W^{l+3} + (K^{(l+3)} - S^{(l+3)})\right) + (K^{(l+2)} - S^{(l+2)})\right) + (K^{(l+1)} - S^{(l+1)}) && (11)\\
&= \left(\prod_{k=l+1}^{L} S^{(k)}\right) W^{(L)} + \sum_{j=l+1}^{L} (K^{(j)} - S^{(j)}) \prod_{i=l+1}^{j-1} S^{(i)} && (12)
\end{aligned}$$

Plugging in $l = 0$ and $W^{(L)} = 1$ gives:

$$\begin{aligned}
W^{(0)} &= \left(\prod_{k=1}^{L} S^{(k)}\right) + \sum_{j=1}^{L}(K^{(j)} - S^{(j)}) \prod_{i=1}^{j-1} S^{(i)} && (13)\\
&= \left(\prod_{k=1}^{L} S^{(k)}\right) + \left(\sum_{j=1}^{L} K^{(j)} \prod_{i=1}^{j-1} S^{(i)}\right) - \left(\sum_{j=1}^{L}\prod_{i=1}^{j} S^{(i)}\right) && (14)\\
&= \left(\sum_{j=1}^{L} K^{(j)} \prod_{i=1}^{j-1} S^{(i)}\right) - \left(\sum_{j=1}^{L-1}\prod_{i=1}^{j} S^{(i)}\right) && (15)\\
&= \left(\sum_{j=1}^{L} K^{(j)} \prod_{i=1}^{j-1} S^{(i)}\right) - \left(\sum_{j=1}^{L}\prod_{i=1}^{j-1} S^{(i)}\right) + 1 && \text{Add empty prod } \prod_{i=1}^{0} S^{(i)} := 1 \text{ to sum} \quad (16)\\
&= \left(\sum_{j=1}^{L}(K^{(j)} - 1) \prod_{i=1}^{j-1} S^{(i)}\right) + 1 && (17)
\end{aligned}$$

For more information about receptive fields, see Araujo, et al., "Computing Receptive Fields of Convolutional Neural Networks", Distill, 2019.