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

Homework 4

This homework is due on Friday, October 3, 2025, at 10:59PM.

1. Newton-Schulz Runtime

Let us consider the Newton-Schulz update for a parameter matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, using degree-3 odd polynomial p:

$$p(W) = \frac{1}{2}\left(3I_{d_{\text{out}}} - WW^T\right)W\,.$$

(a) First, we will analyze the runtime of a single iteration. Assume that the runtime is dominated by matrix multiplication (which is often true for large $d_{\text{out}}, d_{\text{in}}$) and the runtime of multiplying a $n \times m$ matrix by a $m \times p$ matrix takes $cmnp$ runtime. **What is the runtime of each iteration?**

Solution: The cost consists of two matrix multiplications:

- WW^T initially is between a $d_{\text{out}} \times d_{\text{in}}$ and $d_{\text{in}} \times d_{\text{out}}$ matrix which takes $2d_{\text{out}}^2 d_{\text{in}}$ time.
- $(3I_{d_{\text{out}}} - WW^T)W$ at the end is between a $d_{\text{out}} \times d_{\text{out}}$ and $d_{\text{out}} \times d_{\text{in}}$ which takes $2d_{\text{out}}^2 d_{\text{in}}$ time.

So the total runtime is:

$$2cd_{\text{out}}^2 d_{\text{in}}\,.$$

(b) Now, consider the case where $d_{\text{out}} \gg d_{\text{in}}$. **Is there a way to compute $p(W)$ faster? Explain how this can be done and report the updated runtime of each iteration?**
(Hint: Consider the Gram matrix W^TW instead of WW^T. What is the runtime of computing W^TW? Is there a way to rewrite $p(W)$ so that it uses W^TW instead of WW^T?)

Solution: Note that we can rewrite $p(W)$ as:

$$p(W) = \frac{1}{2}\left(3I_{d_{\text{out}}} - WW^T\right)W = \frac{1}{2}W\left(3I_{d_{\text{in}}} - W^TW\right)$$

To show this, we can manipulate the original expression:

$$\begin{aligned}\frac{1}{2}\left(3I_{d_{\text{out}}} - WW^T\right)W &= \frac{1}{2}\left(3W - WW^TW\right)\\ &= \frac{1}{2}\left(3W - W(W^TW)\right)\\ &= \frac{1}{2}W\left(3I_{d_{\text{in}}} - W^TW\right)\end{aligned}$$

where we use associativity of matrix multiplication. Again, the cost of $p(W)$ consists of two matrix multiplications:

- W^TW initially is between a $d_{\text{in}} \times d_{\text{out}}$ and $d_{\text{out}} \times d_{\text{in}}$ matrix which takes $2d_{\text{in}}^2 d_{\text{out}}$ time.
- $W(3I_{d_{\text{in}}} - W^TW)$ at the end is between a $d_{\text{out}} \times d_{\text{in}}$ and $d_{\text{in}} \times d_{\text{in}}$ which takes $2d_{\text{in}}^2 d_{\text{out}}$ time.

So the total runtime is:

$$2cd_{\text{in}}^2 d_{\text{out}}\,,$$

which is better when $d_{\text{out}} \gg d_{\text{in}}$.

2. MuP at the Unit Scale

By now, we have seen how the maximal-update parameterization allows us to properly scale updates to weights, based on the shapes of the dense layers. In last week's homework, we hinted at how this procedure can applied either as a layer-wise learning rate, or as a direct adjustment of the forward pass graph.

This time, we will consider how these same principles can be used in the context of low-precision training. GPUs that support low-precision training tend to exhibit a linear speedup in computation time as precision is lowered (i.e. doing a matrix multiplication in fp16 is twice as fast as in fp32). For this reason, we would like to design training algorithms that can remain numerically stable even at precisions as low as 8 bits.

(a) You are designing a neural network training algorithm that will trained with fp8 parameters (assume activations will be calculated in full precision and we don't need to worry about them). **Why would it make sense for parameters to be initialized from $N(0,1)$ as oppossed to e.g. Xavier initialization?** As a starting point, note that fp8 can only represent 255 possible values, and you can view these values at: https://asawicki.info/articles/fp8_tables.php.

Solution: The number of representable values in fp8 is small, so if all our parameters were initialized at a small scale, we would run into rounding issues. The dynamic range of fp8 is well suited to numbers that roughly follow a unit normal distribution.

(b) We have initialized our parameters from $N(0,1)$. However, we have now lost the desirable properties of Xavier initialization, and our activations are exploding as they propagate deeper into the network. To solve this, we can assign a constant (float) scalar to be multiplied with the activations:

$$\mathbf{y} = cW\mathbf{x}.$$

What should the constant scalar c be to recover the benefits of standard Xavier initialization?

Solution: Normally, Xavier initialization initializes weights from $N(0, 1/\sqrt{d_{\text{in}}})$. To recover this, we need $c = 1/\sqrt{d_{\text{in}}}$.

(c) Now, let us consider an update to the weights ΔW. We would like to properly scale this update, such that the resulting $\Delta\mathbf{y} = c\Delta W x$ is controlled. Assume that $\mathbf{x}$ has an RMS norm of 1. **What should the maximum spectral norm of ΔW be such that $\Delta\mathbf{y}$ has an RMS norm no larger than 1?**

Solution: We must first convert between the spectral norm and the RMS-to-RMS induced norm of ΔW. The RMS-to-RMS induced norm is given by $\sqrt{d_{\text{in}}}/\sqrt{d_{\text{out}}}$ times the spectral norm, therefore an RMS-to-RMS norm of 1 equals a spectral norm of $\sqrt{d_{\text{out}}}/\sqrt{d_{\text{in}}}$. As we inherit the scaling factor $c = 1/\sqrt{d_{\text{in}}}$ from the forward pass, the maximum spectral norm of ΔW is $\sqrt{d_{\text{out}}}$.

(d) Let us consider the case of SignSGD. Assume that our minibatch is of size 1. You saw in discussion that $\text{sign}(\nabla_W L)$ is a rank-1 matrix. **What learning rate α is required to ensure that the overall update of $\alpha \cdot \textbf{sign}(\nabla_W L)$ satisfies the spectral norm constraint from part (c)?** Ensure your answer works on rectangular weight matrices.

Solution: First, note that for a minibatch of size 1, the gradient $\nabla_W L$ can be expressed as an outer product of two vectors (e.g., the gradient with respect to the output g and the input vector x^T), making it a rank-1 matrix. The post-sign matrix is equivalently rank-1, and can be seen as $\text{sign}(gx^T) = \text{sign}(g)\text{sign}(x)^T$.

For rank-1 matrices, the spectral norm is equal to the product of the L2 norms of the two vectors. The L2 norm of $\text{sign}(g)$ is $\sqrt{d_{\text{out}}}$, and the L2 norm of $\text{sign}(x)$ is $\sqrt{d_{\text{in}}}$. Therefore, the spectral norm of $\text{sign}(\nabla_W L)$ is $\sqrt{d_{\text{in}} d_{\text{out}}}$. An alternate way to derive this is to recall that for rank-1 matrices, the spectral norm is equal to the Frobenius norm, which is the square-root of the sum of squares of elements, and is $\sqrt{d_{\text{in}} d_{\text{out}}}$ for $\text{sign}(\nabla_W L)$.

Therefore, to ensure that the overall update $\alpha \cdot \text{sign}(\nabla_W L)$ satisfies the spectral norm constraint from part (c), we need to choose α such that:

$$\alpha \cdot \sqrt{d_{\text{in}} d_{\text{out}}} = \sqrt{d_{\text{out}}}.$$

and therefore,

$$\alpha = 1/\sqrt{d_{\text{in}}}.$$

(e) Let us consider the usage of Muon-style methods to orthogonalize our gradients. Consider the following orthogonalized update rule:

$$U, \Sigma, V^T = \text{SVD}(\nabla_W L) \tag{1}$$

$$\Delta W = \alpha \cdot U V^T. \tag{2}$$

where we use the compact form of the SVD. **What learning rate α is required to ensure that the overall update ΔW satisfies the spectral norm constraint from part (c)?**

Solution: By construction, the matrix UV^T has a spectral norm of 1. Looking at part (c), we see the spectral norm we want for the update and so should multiply by that factor — namely $\alpha = \sqrt{d_{\text{out}}}$.

(f) SignGD, Adam, and Muon share a similar property that the global scale of raw gradients does not affect the final update direction. Now consider the backwards pass of a series of dense layers, where each layer follows the scaled definition from part (b): $\mathbf{x}_{n+1} = c_n W_n \mathbf{x}_n$. You may assume there is no activation function for simplicity. Recall that $\nabla_{x_n} L$ can be recursively calculated from $\nabla_{x_{n+1}} L$. **Is there a setting where the scale of these *intermediate backpropagated gradients* can also be ignored?**

Solution: Yes. As long as the network does not branch (or uses the same scaling at every branch), the global scale of backpropagated gradients can be ignored or adjusted. This can be understood via the chain rule – any constant factors in the intermediate backpropagated gradients can be factored out and applied at the end.

(g) In the setting above, without any adjustments, **will intermediate backpropagated gradients suffer from an explosion or vanishing effect as they are backpropagated?** You may assume that W is rank-1, such that the spectral norm is equal to the Frobenius norm, and each parameter is unit scaled. What constants should the intermediate backpropagated gradients be multiplied by to ensure that they remain stable?

Solution: There will be an explosion in magnitude. Recall the gradient of an activation x_n is given by:

$$\nabla_{x_n} L = c_n W_n^T \nabla_{x_{n+1}} L.$$

Therefore, the inner term $(c_n W_n^T)$ must have a spectral norm of 1 to ensure stability.

From part (b), we know that $c_n = 1/\sqrt{d_{\text{in}}}$. The spectral norm of W_n^T is equal to its Frobenius norm, which for a unit-scaled parameters is $\sqrt{d_{\text{in}} d_{\text{out}}}$. Therefore, the inner term has a spectral norm of $\sqrt{d_{\text{out}}}$, which will lead to a numerical explosion.

To solve this, we can multiply the intermediate backpropagated gradients by $1/\sqrt{d_{\text{out}}}$. This ensures

that the inner term has a spectral norm of 1, and the magnitude of backpropagated gradients remain stable.

3. Understanding Convolution as Finite Impulse Response Filter

For the discrete time signal, the output of linear time invariant system is defined as:

$$y[n] = x[n] * h[n] = \sum_{i=-\infty}^{\infty} x[n-i] \cdot h[i] = \sum_{i=-\infty}^{\infty} x[i] \cdot h[n-i] \tag{3}$$

where x is the input signal, h is impulse response (also referred to as the filter). Please note that the convolution operations is to 'flip and drag'. But for neural networks, we simply implement the convolutional layer without flipping and such operation is called correlation. Interestingly, in CNN those two operations are equivalent because filter weights are initialized and updated. Even though you implement 'true' convolution, you just ended up with getting the flipped kernel. **In this question, we will follow the definition in 3.**

Now let's consider rectangular signal with the length of L (sometimes also called the "rect" for short, or, alternatively, the "boxcar" signal). This signal is defined as:

$$x(n) = \begin{cases} 1 & n = 0, 1, 2, ..., L-1 \\ 0 & \text{otherwise} \end{cases}$$

Here's an example plot for $L = 7$, with time indices shown from -2 to 8 (so some implicit zeros are shown):

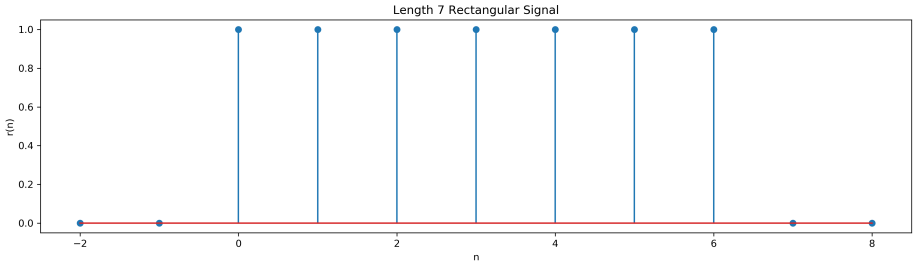

Figure 1: The rectangular signal with the length of 7

(a) The impulse response is define as:

$$h(n) = (\frac{1}{2})^n u(n) = \begin{cases} (\frac{1}{2})^n & n = 0, 1, 2, ... \\ 0 & \text{otherwise} \end{cases}$$

Compute and plot the convolution of $x(n)$ and $h(n)$. For illustrative purposes, your plot should start at -6 and end at +12.

Solution: Let's divide this problem into three cases: 1) $n < 0$, 2) $0 \leq n < L-1$ and $n \geq L-1$. For the first case, $x(i) = 0$ and $h[n-i] = 0$ if $i < 0$, we only need to consider $i \geq 0$.

$$y(n) = \sum_{i=-\infty}^{\infty} x[i] \cdot h[n-i]$$

$$= \sum_{i=0}^{L-1} h[n-i]$$

Here, $n-i$ is negative, $h(n-i)=0$,

$$y(n) = \sum_{i=0}^{L-1} h(n-i) = 0$$

Consider the second case: $0 \le n < L-1$. Since for $n-i<0$ $h(n)=0$, we only need to consider $i \le n$

$$\begin{aligned} y(n) &= \sum_{i=-\infty}^{\infty} x[i] \cdot h[n-i] \\ &= \sum_{i=0}^{n} h[n-i] \\ &= \sum_{i=0}^{n} \frac{1}{2^i} \\ &= 2 - \frac{1}{2^n} \end{aligned}$$

For the last case, $x(n)=0$ when $n \ge L$. Therefore,

$$\begin{aligned} y(n) &= \sum_{i=-\infty}^{\infty} x[i] \cdot h[n-i] \\ &= \sum_{i=0}^{L-1} h[n-i] \\ &= \sum_{i=0}^{L-1} \frac{1}{2^i} \\ &= \frac{2^L - 1}{2^n} \end{aligned}$$

So $y(n)$ is

$$y(n) = \begin{cases} 0 & n < 0 \\ 2 - \frac{1}{2^n} & 0 \le n < L-1 \\ \frac{2^L - 1}{2^n} & \text{otherwise} \end{cases} \tag{4}$$

(b) Now let's shift $x(n)$ by N, i.e. $x_2(n) = x(n-N)$. Let's put $N=5$ **Then, compute** $y_2(n) = h(n) * x_2(n)$**. Which property of the convolution can you find?**

Solution: The output is just shift the result of Eq. (4) by N. Therefore the output of this convolution is $y[n-N] = y[n-5]$. We can observe translational invariance (shift invariance / equivariance) property of the convolution operation.

Now, let's extend 1D to 2D. The example of 2D signal is the image. The operation of 2D convolution

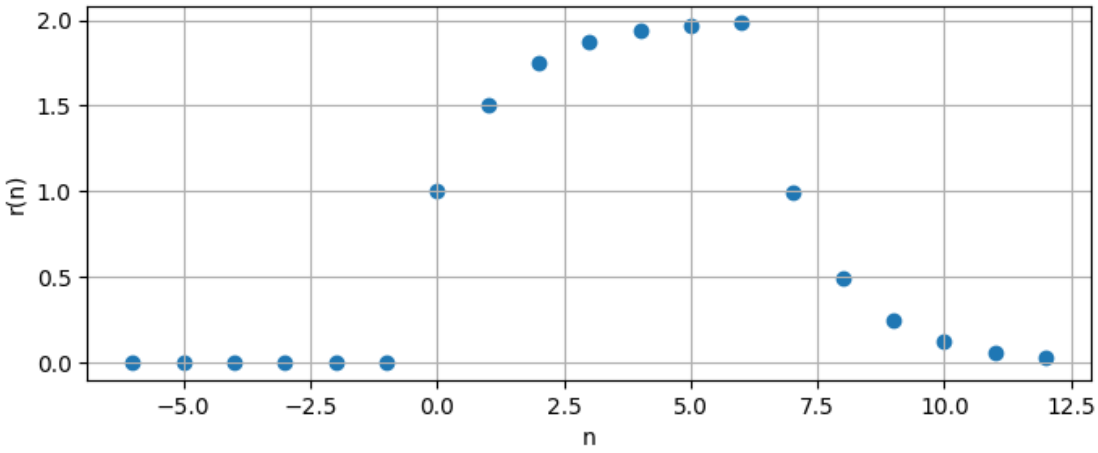

Figure 2: The rectangular signal with the length of 7 after convolution

is defined as follows:

$$y[m,n] = x[m,n] * h[m,n] = \sum_{i,j=-\infty}^{\infty} x[m-i, n-j] \cdot h[i,j] = \sum_{i,j=-\infty}^{\infty} x[i,j] \cdot h[m-i, n-j] \tag{5}$$

, where x is input signal, h is FIR filter and y is the output signal.

(c) 2D matrices, x and h are given like below:

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix} \tag{6}$$

$$h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{7}$$

Then, evaluate y. Assume that there is no pad and stride is 1.

Solution:

$$\begin{bmatrix} -40 & -40 & -40 \\ -40 & -40 & -40 \\ -40 & -40 & -40 \end{bmatrix}$$

(d) Now let's consider striding and padding. Evaluate y for following cases:

i. stride, pad = 1, 1

ii. stride, pad = 2, 1

Solution:

i. On padding with 1, x becomes $\hat{x}$

$$\hat{x} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 9 & 10 & 0 \\ 0 & 11 & 12 & 13 & 14 & 15 & 0 \\ 0 & 16 & 17 & 18 & 19 & 20 & 0 \\ 0 & 21 & 22 & 23 & 24 & 25 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8}$$

Applying $y = \hat{x} * h$

$$y = \begin{bmatrix} -19 & -28 & -32 & -36 & -29 \\ -30 & -40 & -40 & -40 & -30 \\ -30 & -40 & -40 & -40 & -30 \\ -30 & -40 & -40 & -40 & -30 \\ 49 & 68 & 72 & 76 & 59 \end{bmatrix} \tag{9}$$

ii. Applying convolution with stride 2 on $\hat{x}$:

$$y = \begin{bmatrix} -19 & -32 & -29 \\ -30 & -40 & -30 \\ 49 & 72 & 59 \end{bmatrix} \tag{10}$$

4. Feature Dimensions of Convolutional Neural Network

In this problem, we compute output feature shape of convolutional layers and pooling layers, which are building blocks of CNN. Let's assume that input feature shape is $W \times H \times C$, where W is the width, H is the height and C is the number of channels of input feature.

(a) A convolutional layer has 4 architectural hyperparameters: the filter size(K), the padding size (P), the stride step size (S) and the number of filters (F). **How many weights and biases are in this convolutional layer? And what is the shape of output feature that this convolutional layer produces?**

Solution:

The number of weights $= K^2CF$
The number of biases $= F$
$W' = \lfloor (W - K + 2P)/S \rfloor + 1$
$H' = \lfloor (H - K + 2P)/S \rfloor + 1$
$C' = F$

(b) A max pooling layer has 2 architectural hyperparameters: the stride step size(S) and the "filter size" (K). **What is the output feature shape that this pooling layer produces?**

Solution:

$W' = (W - K)/S + 1$
$H' = (H - K)/S + 1$
$C' = C$

(c) Let's assume that we have the CNN model which consists of L successive convolutional layers and the

filter size is K and the stride step size is 1 for every convolutional layer. Then **what is the receptive field size of the last output**?

Solution:

$L * (K-1) + 1 = O(LK)$

Note that, the receptive field size increases linearly as we add more layers

We also accept $(L-1) * (K-1) + 1 = O(LK)$ as the answer.

(d) Consider a downsampling layer (e.g. pooling layer and strided convolution layer). In this problem, we investigate pros and cons of downsampling layers. This layer reduces the output feature resolution and this implies that the output features lose a certain amount of spatial information. Therefore when we design CNNs, we usually increase tchannel length to compensate this loss. For example, if we apply a max pooling layer with a kernel size of 2 and a stride of 2, we increase the output feature size by a factor of 2. **If we apply this max pooling layer, how much does the receptive field increases? Explain the advantage of decreasing the output feature resolution with the perspective of reducing the amount of computation.**

Solution: The receptive field size increases by a factor of 2. Since the spatial resolution decrease by 1/4 and the channel size increases by 2, the feature size decreases by a factor of 2. Therefore, the number of MAC operations after the pooling layer decreases.

(e) Let's take a real example. We are going to describe a convolutional neural net using the following pieces:

- CONV3-F denotes a convolutional layer with F different filters, each of size $3 \times 3 \times C$, where C is the depth (i.e. number of channels) of the activations from the previous layer. Padding is 1, and stride is 1.
- POOL2 denotes a 2×2 max-pooling layer with stride 2 (pad 0)
- FLATTEN just turns whatever shape input tensor into a one-dimensional array with the same values in it.
- FC-K denotes a fully-connected layer with K output neurons.

Note: All CONV3-F and FC-K layers have biases as well as weights. **Do not forget the biases when counting parameters.**

Now, we are going to use this network to do inference on a single input. **Fill in the missing entries in this table of the size of the activations at each layer, and the number of parameters at each layer. You can/should write your answer as a computation (e.g.** $128 \times 128 \times 3$**) in the style of the already filled-in entries of the table.**

Layer	Number of Parameters	Dimension of Activations
Input	0	$28 \times 28 \times 1$
CONV3-10	**Solution:** $3 \times 3 \times 1 \times 10 + 10$	$28 \times 28 \times 10$
POOL2	0	$14 \times 14 \times 10$
CONV3-10	$3 \times 3 \times 10 \times 10 + 10$	**Solution:** $14 \times 14 \times 10$
POOL2	**Solution:** 0	**Solution:** $7 \times 7 \times 10$
FLATTEN	0	490
FC-3	**Solution:** $490 \times 3 + 3$	3

(f) Consider a new architecture:

CONV2-3 $\rightarrow$ ReLU $\rightarrow$ CONV2-3 $\rightarrow$ ReLU $\rightarrow$ GAP (Global Average Pool) $\rightarrow$ FC-3

Each CONV2-3 layer has stride of 1 and padding of 1. Note that we use **circular padding** (i.e. wrap-around) for this task. Instead of using zeros, circular padding makes it as though the virtual column before the first column is the last column and the virtual row before the first row is the last row — treating the image as though it was on a torus.

Here, the GAP layer is an average pooling layer that computes the per-channel means over the entire input image.

You are told the behavior for an input image with a horizontal edge, $\mathbf{x}_1$ and an image with a vertical edge, $\mathbf{x}_2$:

$$\mathbf{x}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{x}_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Suppose we knew that the GAP output features when fed $\mathbf{x}_1$ and $\mathbf{x}_2$ are

$$\mathbf{g}_1 = f(\mathbf{x}_1) = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{g}_2 = f(\mathbf{x}_2) = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}$$

Use what you know about the invariances/equivariances of convolutional nets to compute the $\mathbf{g}_i$ corresponding to the following $\mathbf{x}_i$ images.

- $\mathbf{x}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$ **Solution:** $\mathbf{g}_3 = f(\mathbf{x}_3) = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}$

- $\mathbf{x}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ **Solution:** $\mathbf{g}_4 = f(\mathbf{x}_4) = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}$

Solution: In both cases, this is because the input is just a circular shift of the examples earlier and we know that this CNN is invariant to circular shifts. It is important to note that this only works because we use cicular padding. If we are to use other padding method like zero padding, the network would not be necessarily invariant to circular shift of examples.

5. Desigining 2D Filter (Coding Question)

Convolutional layer, which is the most important building block of CNN, actively utilizes the concept of filters used in traditional image processing. Therefore, it is quite important to know and understand the types and operation of image filters.

Look at HandDesignFilters.ipynb. In this notebook, we will design two convolution filters by hand to understand the operation of convolution:

(a) **Blurring filter.**

(b) **Edge detection filter.**

For each type of filter, please **include the image generated by the Jupyter Notebook in your submission to the written assignment**. The image should be added to the PDF document that you will be submitting.

6. Inductive Bias of CNNs (Coding Question)

In this problem, you will follow the edge_detection.ipynb notebook to understand the inductive bias of CNNs.

1 Overfitting Models to Small Dataset: **Fill out notebook section (Q1).**

(i) Can you find any interesting patterns in the learned filters?

Solution: Somewhat looks like edge detection filters. (Every reasonable answer is correct)

2 Sweeping the Number of Training Images: **Fill out notebook section (Q2).**

(i) Compare the learned kernels, untrained kernels, and edge-detector kernels. What do you observe?

Solution: When CNN's performance is low, Kernels look almost random. Interestingly, with the dumb luck of initialization, there are some kernels that already look like edge detectors. When CNN's performance is high, Kernels look like edge detectors. (Every reasonable answer is correct)

(ii) We freeze the convolutional layer and train only final layer (classifier). In a high data regime, the performance of CNN initialized with edge detectors is worse than CNN initialized with random weights. Why do you think this happens?

Solution: As mentioned earlier, inductive bias benefits both performance and training efficiency. But not always that's the case. The training result can be worse if we inject too much into the training process/model or incorrectly. We can understand it through the lens of bias-variance tradeoff. If we introduce inductive biases to the model or training process, it increases bias while decreasing variance. But note that a large number of data also reduce the variance. The edge detection problem is so simple that 50 images per class are large enough to decrease the decent amount of variance. But we inject too many inductive biases into the model: initializing with edge detection filters and freezing them. Therefore, the bias term dominates the generalization error in the high data regime. For more information, please refer to this paper.

3 Checking the Training Procedures: **Fill out notebook section (Q3).**

(i) List every epochs that you trained the model. Final accuracy of CNN should be at least 90% for 10 images per class.

Solution: There is no definite answer in this question. If you achieve 90% for 20 images per class with CNN model, your answer is correct

(ii) Check the learned kernels. What do you observe?

Solution: Learned kernels look like edge detectors. (Every reasonable answer is correct)

(iii) (optional) You might find that with the high number of epochs, validation loss of MLP is increasing whild validation accuracy increasing. How can we interpret this?

Solution: As the model overfits to training set, the model becomes overconfident.

(iv) (optional) Do hyperparameter tuning. And list the best hyperparameter setting that you found and report the final accuracy of CNN and MLP.

Solution: There is no definite answer in this question. If you find any hyperparameter configuration that performs better than the given, your answer is correct.

(v) How much more data is needed for MLP to get a competitive performance with CNN?

Solution: Any number you found is correct if the validation accuracy of MLP is similar to that of CNN. Considering the data generating process, we can find four significant variables for this dataset: 1) edge location, 2) edge width, 3) edge intensity and 4) background intensity. The first two variables mainly determine the patterns of images. As you can see in the data generation code, we randomly sample from [1, 2, 3, 4, 5] as the edge width and [1 $\sim$ 28] as the edge location for each edge. Hence there are 280 possible cases. CNN achieves almost 100% validation accuracy with about 50 training images, but MLP needs a much larger number of data points to achieve matched validation accuracy.

4 Domain Shift between Training and Validation Set: **Fill out notebook section (Q4).**

(i) Why do you think the confusion matrix looks like this?

Solution: You may see different confusion matrices depending on the random initialization seeds. Any reasonable answer is correct. One example – "CNN overfits to the edges that are located in the left half or upper half of the image. As a result, it is more likely to classify images in validatation set with edges that are located in the right half or lower half of the image as the class of images with no edge. However, MLP overfits to the edges that are located in the fraction of edges that are located in the left upper part and right lower part of the image. This causes the MLP to wrongly classify the images with vertical edge to the class of images with horizontal edge and vice versa."

(ii) Why do you think MLP fails to learn the task while CNN can learn the task? (Hint: Think about the model architecture.)

Solution: CNN is translational invariant. Therefore, CNN can learn the task even though the edges are located in the different half of the image. However, MLP is not translational invariant. Therefore, MLP fails to learn the task.

5 When CNN is Worse than MLP: **Fill out notebook section (Q5).**

(i) What do you observe? What is the reason that CNN is worse than MLP? (Hint: Think about the model architecture.)

Solution: CNN utilizes the local correlation of the image. However, the local correlation is destroyed by the random permutation. Therefore, CNN fails to learn the task. However, MLP is not affected by the random permutation because it is 'fully connected'. Therefore, MLP can learn the task.

(ii) Assuming we are increasing kernel size of CNN. Does the validation accuracy increase or decrease? Why?

Solution: The validation accuracy will increase. This is because as kernel gets larger, convolutional layer becomes more 'fully connected'. Therefore, CNN can learn the task even though the local correlation is destroyed by the random permutation.

(iii) How do the learned kernels look like? Explain why.

Solution: The learned kernels are not that different from the randomly initialized kernels. This is because CNN cannot capture the local correlation of the image. Recalling backpropagation of CNN, CNN kernels are optimized in the way that maximizes the correlation of grids of images. (Any reasonable answer is also correct)

6 Increasing the Number of Classes: **Fill out notebook section (Q6).**

(i) Compare the performance of CNN with max pooling and average pooling. What are the advantages of each pooling method?

Solution: With <100 training images, max pooling performs better than average pooling. With 100 training image, average pooling is on par with max pooling.
Average pooling method smooths out the image and hence the sharp features may not be identified when this pooling method is used. Max pooling selects the brighter pixels from the image. It is useful when the background of the image is dark and we are interested in only the lighter pixels of the image. We cannot say that a particular pooling method is better over other generally. The choice of pooling operation is made based on the data at hand.

7. Weights and Gradients in a CNN

In this homework assignment, we aim to accomplish two objectives. Firstly, we seek to comprehend that the weights of a CNN are a weighted average of the images in the dataset. This understanding is crucial in answering a commonly asked question: does a CNN memorize images during the training process? Additionally, we will analyze the impact of spatial weight sharing in convolution layers. Secondly, we aim to gain an understanding of the behavior of max-pooling and avg-pooling in backpropagation. By accomplishing these objectives, we will enhance our knowledge of CNNs and their functioning.

Let's consider a convolution layer with input matrix $\mathbf{X} \in \mathbb{R}^{n \times n}$,

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix}, \tag{11}$$

weight matrix $\mathbf{w} \in \mathbb{R}^{k \times k}$,

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,k} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1} & w_{k,2} & \cdots & w_{k,k} \end{bmatrix}, \tag{12}$$

and output matrix $\mathbf{Y} \in \mathbb{R}^{m \times m}$,

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,m} \end{bmatrix}. \tag{13}$$

For simplicity, we assume the number of the input channel (of $\mathbf{X}$ is) and the number of the output channel (of output $\mathbf{Y}$) are both 1, and the convolutional layer has no padding and a stride of 1.

Then for all i, j,

$$y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}, \tag{14}$$

or

$$\mathbf{Y} = \mathbf{X} * \mathbf{w}, \tag{15}$$

, where $*$ refers to the convolution operation. For simplicity, we omitted the bias term in this question.

Suppose the final loss is $\mathcal{L}$, and the upstream gradient is $d\mathbf{Y} \in \mathbb{R}^{m,m}$,

$$d\mathbf{Y} = \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix}, \tag{16}$$

where $dy_{i,j}$ denotes $\dfrac{\partial \mathcal{L}}{\partial y_{i,j}}$.

(a) **Derive the gradient to the weight matrix $d\mathbf{w} \in \mathbb{R}^{k,k}$,**

$$d\mathbf{w} = \begin{bmatrix} dw_{1,1} & dw_{1,2} & \cdots & dw_{1,k} \\ dw_{2,1} & dw_{2,2} & \cdots & dw_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ dw_{k,1} & dw_{k,2} & \cdots & dw_{k,k} \end{bmatrix}, \tag{17}$$

where $dw_{h,l}$ denotes $\dfrac{\partial \mathcal{L}}{\partial w_{h,l}}$. Also, **derive the weight after one SGD step with a batch of a single image**.

Solution:

The forward propagation rule is

$$y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}. \tag{18}$$

Use chain rule

$$\frac{\partial \mathcal{L}}{\partial w_{h,l}} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{h,l}}, \tag{19}$$

and

$$\frac{\partial y_{i,j}}{\partial w_{h,l}} = x_{i+h-1,j+l-1}, \tag{20}$$

so we have

$$dw_{h,l} = \sum_{i=1}^{m} \sum_{j=1}^{m} x_{i+h-1,j+l-1} dy_{i,j} \tag{21}$$

$$d\mathbf{w} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix} * \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix} \tag{22}$$

$$= \mathbf{X} * d\mathbf{Y}. \tag{23}$$

For one SGD step with learning rate η, suppose the input and output are $\mathbf{X}$ and $\mathbf{Y}$, then

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta d\mathbf{w} = \mathbf{w_t} - \eta \mathbf{X} * d\mathbf{Y} \tag{24}$$

We can conclude that **during the training of CNN weights with SGD, the weights of a CNN are a weighted average of images patches in the dataset**. This is because the gradient, which is a weighted average of image patches in the dataset, is used to update the weights in SGD, so the trained weight is the initial weight plus a weighted average of the gradients.

(b) The objective of this part is to investigate the effect of spatial weight sharing in convolution layers on the behavior of gradient norms with respect to changes in image size.

For simplicity of analysis, we assume $x_{i,j}, dy_{i,j}$ are independent random variables, where for all i, j:

$$\mathbb{E}[x_{i,j}] = 0, \tag{25}$$

$$\mathrm{Var}(x_{i,j}) = \sigma_x^2, \tag{26}$$

$$\mathbb{E}[dy_{i,j}] = 0, \tag{27}$$

$$\mathrm{Var}(dy_{i,j}) = \sigma_g^2. \tag{28}$$

Derive the mean and variance of $dw_{h,l} = \dfrac{\partial \mathcal{L}}{\partial W_{h,l}}$ for each i, j a function of n, k, σ_x, σ_g. What is the asymptotic growth rate of the standard deviation of the gradient on $dw_{h,l}$ with respect to the length and width of the image n?

Hint: there should be no m in your solution because m can be derived from n and k.

Hint: you cannot assume that $x_{i,j}$ and $dy_{i,j}$ follow normal distributions in your derivation or proof.

Solution:

From the previous part, for each i, j, we have

$$dw_{i,j} = \sum_{h=1}^{m}\sum_{l=1}^{m} dy_{h,l} x_{i+h-1,j+l-1}. \tag{29}$$

Given the assumption that $dy_{h,l}$ and $x_{i,j}$ are independent random variables, we have $\mathbb{E}[dy_{h,l}x_{i,j}] = \mathbb{E}[dy_{h,l}]\mathbb{E}[x_{i,j}]$ for every i, j, h, l, so:

$$\begin{aligned}
\mathbb{E}[dw_{i,j}] &= \sum_{h=1}^{m}\sum_{l=1}^{m} \mathbb{E}[dy_{h,l}]\mathbb{E}[x_{i+h-1,j+l-1}] && (30)\\
&= 0. && (31)
\end{aligned}$$

As for the variance, similarly, we have $\mathbb{E}[dy_{h,l}^2 x_{i,j}^2] = \mathbb{E}[dy_{h,l}^2]\mathbb{E}[x_{i,j}^2]$ for every i, j, h, l, so:

$$\begin{aligned}
\mathrm{Var}(dw_{i,j}) &= \mathbb{E}[dw_{i,j}^2] - \mathbb{E}[dw_{i,j}]^2 && (32)\\
&= \mathbb{E}[dw_{i,j}^2] - 0 && (33)\\
&= \mathbb{E}\left[\left(\sum_{h=1}^{m}\sum_{l=1}^{m} dy_{h,l} x_{i+h-1,j+l-1}\right)^2\right] && (34)\\
&= \sum_{h=1}^{m}\sum_{l=1}^{m} \mathbb{E}[dy_{h,l}^2]\mathbb{E}[x_{i+h-1,j+l-1}^2] && (35)\\
&= \sum_{h=1}^{m}\sum_{l=1}^{m} (\mathrm{Var}(dy_{h,l}) + \mathbb{E}[dy_{h,l}]^2)(\mathrm{Var}(x_{i+h-1,j+l-1}) + \mathbb{E}[x_{i+h-1,j+l-1}]^2) && (36)\\
&= \sum_{h=1}^{m}\sum_{l=1}^{m} \mathrm{Var}(dy_{h,l})\mathrm{Var}(x_{i+h-1,j+l-1}) && (37)\\
&= m^2\sigma_x^2\sigma_g^2. && (38)
\end{aligned}$$

Because there is no padding and stride is 1, we have $m = n - k + 1$, so

$$\mathrm{Std}(dw_{i,j}) = \sqrt{\mathrm{Var}(dw_{i,j})} = \sqrt{(n-k+1)^2\sigma_x^2\sigma_g^2} = \Theta(n), \tag{39}$$

i.e., the stardard variance of the gradient on each parameter grows linearly as the image size increases.

(c) **For a network with only 2x2 max-pooling layers (no convolution layers, no activations), what will be $d\mathbf{X} = [dx_{i,j}] = [\frac{\partial \mathcal{L}}{\partial x_{i,j}}]$? For a network with only 2x2 average-pooling layers (no convolution layers, no activations), what will be $d\mathbf{X}$?**

HINT: Start with the simplest case first, where $\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$. *Further assume that top left value is selected by the max operation. i.e.*

$$y_{1,1} = x_{1,1} = \max(x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}) \tag{40}$$

Then generalize to higher dimension and arbitrary max positions.

Solution:

In the simplest case, output $\mathbf{Y}$ has size 1x1. For the max pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } (i,j) = 1,1 \\ 0, & \text{otherwise} \end{cases} \tag{41}$$

Combining all four partial derivatives, we have

$$d\mathbf{X} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} dy_{11} & 0 \\ 0 & 0 \end{bmatrix} \tag{42}$$

For the average pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = 1/4 \tag{43}$$

$$d\mathbf{X} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \\ \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \end{bmatrix} \tag{44}$$

In the general setting, for max pooling, we can notice that x and y have a one to one mapping. Each x value is involved in calculation of exactly one y value. Let $k = i//2$, $l = j//2$, where $//$ performs the floordiv operation, let

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } x_{i,j} = \max(x_{i,j}, x_{i+1,j}, x_{i,j+1}, x_{i+1,j+1}) \\ 0, & \text{otherwise} \end{cases} \tag{45}$$

, then

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} \tag{46}$$

$d\mathbf{X}$ is the matrix contructed by each $dx_{i,j}$. It will have similar pattern to the simplest case. For each 2x2 block, only one of the input pixel has gradient of magnitude one flowing back, and the rest three inputs have zero gradient.

For the average pooling general case,

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \frac{1}{4} \tag{47}$$

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} = \frac{dy_{kl}}{4} \tag{48}$$

Average pooling distributes the gradient evenly across each 2x2 input blocks.

(d) Following the previous part, **discuss the advantages of max pooling and average pooling** in your own words.

Hint: you may find it helpful to finish the question "Inductive Bias of CNNs (Coding Question)" before working on this question

Solution: In general, it depends on the specific problem setting to use max pooling or average pooling.

Advantages of max pooling: it learns invariant features. Max pooling has the advantage of retaining strong features and highlighting the presence of an object in the image.

When average pooling is needed: it is used to reduce the spatial dimensions of the feature map while retaining some information about the distribution of values in the region. Average pooling is better at capturing smooth transitions or subtle changes in the feature map. For example, average pooling usually follows the last convolution layer of feature learning.

8. 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:

- Joey Hong.
- Anant Sahai.
- Kevin Frans.
- Suhong Moon.
- Dominic Carrano.
- Babak Ayazifar.
- Sukrit Arora.

- Romil Bhardwaj.
- Fei-Fei Li.
- Sheng Shen.
- Jake Austin.
- Kevin Li.
- Kumar Krishna Agrawal.
- Peter Wang.
- Qiyang Li.
- Linyuan Gong.
- Long He.

