EECS 182 Deep Neural Networks
Fall 2026 Gireeja Ranade

Homework 5

This homework is due on Wednesday, October 7, 2026, at 10:59PM.

1. Maximal Update Parameterization Research

This homework problem talks about the research papers behind "maximal update parameterization," also called μP in the community. This idea is discussed in detail in the paper Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer. Understanding the full paper is out of scope of the class and requires sophistication that is not a prerequisite, but we want to highlight some points.

(a) Look at Figure 1 of the paper. **What are the two side-by-side figures plotting? What is the observation you make about the width of a matrix?**

Solution: Both figures plot the training loss of models of different hidden layer widths vs the log of the learning rate used for training with Adam. Larger width matrices lead to better performance when using the muP scaling.

(b) Look at Table 3 in the paper. In lecture, we gave an elementary argument to support the $1/\text{fan}_{\text{in}}$ scaling for Adam LR for the hidden layers, the rightmost entry in the bottom row of the paper. **What does the table say is the corresponding standard parameterization?**

Solution: The corresponding standard parameterization is 1.

(c) The paper A Spectral Condition for Feature Learning presents the same scaling using elementary linear algebra. While understanding the entirely of the paper is out of scope of the class, we presented some simplified arguments in lecture.

Look at Desideratum 1. **Why is this behavior desired for feature learning? Can you rewrite this Desideratum in terms of the RMS norm? Similarly, rewrite Condition 1 in terms of the RMS norm as well. How does Condition 1 imply that $\|h_\ell(x)\|_2 \leq \Theta(\sqrt{n_\ell})$ and that $\|\Delta h_\ell(x)\|_2 \leq \Theta(\sqrt{n_\ell})$? What is the key assumption that allows us to also get the lower bounds on $\|h_\ell(x)\|_2$ and $\|\Delta h_\ell(x)\|_2$?**

Solution: This behavior is desired to so that each element of the hidden vector moves by $O(1)$ and maintains a size of $O(1)$ during training. It is good for features to move significantly since we believe that the initialization is far from the optimal point in the loss landscape.

Condition 1 gives:

$$O\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) = ||W||_2 = \sqrt{\frac{n_\ell}{n_{\ell-1}}}||W||_{\text{RMS}\to\text{RMS}} \Rightarrow ||W||_{\text{RMS}\to\text{RMS}} = O(1). \tag{1}$$

$$\|h_\ell(x)\|_2 = \|W_\ell h_{\ell-1}(x)\|_2 \leq ||W_\ell||_2||h_{\ell-1}(x)||_2 \leq O\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) O(\sqrt{n_{\ell-1}}) = O(\sqrt{n_\ell}). \tag{2}$$

$$\|\Delta h_\ell(x)\|_2 = \|W_\ell \Delta h_{\ell-1}(x) + \Delta W_\ell h_{\ell-1}(x) + \Delta W_\ell \Delta h_{\ell-1}(x)\|_2 \tag{3}$$

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

$$\leq \|W_\ell \Delta h_{\ell-1}(x)\|_2 + \|\Delta W_\ell h_{\ell-1}(x)\|_2 + \|\Delta W_\ell \Delta h_{\ell-1}(x)\|_2 \tag{4}$$

$$\leq \|W_\ell\|_2 \|\Delta h_{\ell-1}(x)\|_2 + \|\Delta W_\ell\|_2 \|h_{\ell-1}(x)\|_2 + \|\Delta W_\ell\|_2 \|\Delta h_{\ell-1}(x)\|_2 \tag{5}$$

$$= O\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) O(\sqrt{n_{\ell-1}}) + O\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) O(\sqrt{n_{\ell-1}}) + O\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) O(\sqrt{n_{\ell-1}}) \tag{6}$$

$$= O(\sqrt{n_\ell}). \tag{7}$$

The key assumptions for the lower bounds are Claim 1 and Claim 2 in the paper. Informally, they state that the hidden weight matrices and their updates scale incoming vectors by their spectral norm.

2. Visualizing Maximal Update Parameterization (Coding Question)

In this question, using simple MLPs, you will explore how the maximal update parameterization preserves feature learning as the width of the network changes. Please follow the instructions in this Google Colab notebook.[1]

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

[1] This notebook is very newly developed and has not been as extensively tested as previously assigned coding problems. Nonetheless, we believe it is a great aid in learning these timely concepts. If you encounter any issues, please post on Ed or notify the course staff in their office hours.

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

(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 have already seen in an earlier problem that $\text{sign}(\nabla_W L)$ is a rank-1 matrix. **What learning rate α is required to ensure that the overall update of $\alpha \cdot \text{sign}(\nabla_W L)$ 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:

$$U, \Sigma, V^T = \text{SVD}(\nabla_W L) \tag{8}$$

$$\Delta W = \alpha \cdot UV^T. \tag{9}$$

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

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

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.

4. Understanding Convolution as Finite Impulse Response Filter

In this question, we use mathematical convolution: flip the filter before sliding it over the input. Neural-network libraries typically implement cross-correlation without flipping; either convention can be used for learned filters, but here follow the convolution definition below.

An image is an example of a 2D signal. The operation of 2D convolution is defined as follows:

$$y[m,n] = x[m,n] * h[m,n] = \sum_{i,j=-\infty}^{\infty} x[m-i, n-j] \cdot h[i,j] = \sum_{i,j=-\infty}^{\infty} x[i,j] \cdot h[m-i, n-j] \quad (10)$$

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

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

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

$$h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (12)$$

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

Solution:

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

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

(b) 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 $\hat{x}$

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

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

$$y = \begin{bmatrix} -19 & -28 & -32 & -36 & -29 \\ -30 & -40 & -40 & -40 & -30 \\ -30 & -40 & -40 & -40 & -30 \\ -30 & -40 & -40 & -40 & -30 \\ 49 & 68 & 72 & 76 & 59 \end{bmatrix} \tag{14}$$

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

$$y = \begin{bmatrix} -19 & -32 & -29 \\ -30 & -40 & -30 \\ 49 & 72 & 59 \end{bmatrix} \tag{15}$$

5. 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?**

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

Solution:
$W' = \lfloor (W - K)/S \rfloor + 1$
$H' = \lfloor (H - K)/S \rfloor + 1$
$C' = C$

(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

(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 the channel 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 number of output channels by a factor of 2. **If we apply this max pooling layer, how much does the receptive field increase? Explain the advantage of decreasing the output feature resolution with the perspective of reducing the amount of computation.**

Solution: Let r be the receptive field side length of the features entering the pooling layer, and j the distance, in input pixels, between neighboring features. A 2×2, stride-2 pooling window combines two neighbors per dimension, so it grows the receptive field only from r to $r + j$ (e.g. from 3×3 to 4×4 right after a single 3×3 convolution). Its main effect is that it doubles the spacing to $2j$, so every later layer grows the receptive field twice as fast as before: a 3×3 convolution after the pooling adds $2 \cdot 2j$ instead of $2j$ (for example $3 \to 4 \to 8$, compared with $3 \to 5$ without pooling). Answers stating that the receptive fields of subsequent layers grow twice as fast (roughly doubling) are correct.

For computation: pooling itself keeps the number of channels, and the doubling of channels happens in the next convolutional layer. The number of spatial positions drops to $1/4$, and with the channel count doubled the feature map holds half as many values. A multiply–accumulate (MAC) operation multiplies two numbers and adds the result to a running sum. A convolutional layer's MAC count is proportional to (number of spatial positions) $\times K^2 \times$ (input channels) $\times$ (output channels). Compared with running the same doubled-channel layer at full resolution, the layer after pooling needs $4\times$ fewer MACs. Put differently, doubling both the input and output channels multiplies the MACs by 4 and the $4\times$ fewer positions cancel that, so the network can grow wider and see a larger receptive field without increasing the per-layer computation.

(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.**

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

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:

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

Each CONV2-3 layer has stride of 1 and padding of 1, applied only above the image and to its left, so its output has the same size as its input. 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.**

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

- $\mathbf{x}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$ **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 circular padding. If we are to use other padding method like zero padding, the network would not be necessarily invariant to circular shift of examples.

6. Designing 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.** **Solution:** An $n \times n$ averaging (box) filter has every entry equal to $1/n^2$, so its entries sum to 1 and it replaces each pixel by the mean of its $n \times n$ neighborhood:

```
kernel = np.ones((filter_size, filter_size)) / (filter_size *
                                filter_size)
```

The images should get progressively blurrier as the kernel grows from 3×3 to 6×6 to 9×9: fine texture such as the fox's fur is smoothed away first, while large shapes stay visible. Other low-pass filters (e.g. a Gaussian kernel) are also acceptable.

(b) **Edge detection filter.** **Solution:** The 3×3 discrete Laplacian kernel given in the notebook:

```
kernel = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
```

Its entries sum to 0, so it outputs zero on regions of constant brightness and responds strongly where the brightness changes abruptly. The output image is mostly dark, with bright responses along the outline of the fox and other sharp intensity changes. Other edge detectors (e.g. Sobel kernels) are also acceptable.

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.

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

7. Convolutional Neural Networks

(a) Consider a convolutional neural network layer with:

- Input shape: [10, 3, 32, 32] (batch size, number of input channels, input height, input width)
- Number of filters: 64

For each of the following configurations of kernel size, stride, and padding, **calculate the output dimensions** $[n, c, h, w]$ where n is the batch size, c is the number of channels, and h, w are the output height and width. Assume that each layer has the same input shape and number of filters as specified above.

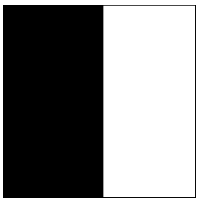
i. Kernel size: 3x3, stride: 1, padding: 1

Solution: Since the output dimensions are calculated as $d_{out} = \lfloor \frac{d_{in}+2p-k}{s} \rfloor + 1$ where d_{in} is the input dimension, d_{out} is the output dimension, p is the padding, k is the kernel size, and s is the stride, we can substitute these values to get the output dimension: $d_{out} = \lfloor \frac{32+2(1)-3}{1} \rfloor + 1 = 32$. The output shape would be $[10, 64, 32, 32]$

ii. Kernel size: 4x4, stride: 2, padding: 0

Solution: Using the same formula above, we get the output shape $[10, 64, 15, 15]$

(b) **Design a 3x3 filter that detects vertical edges like the one shown in the image below.**



$$\begin{bmatrix} _ & _ & _ \\ _ & _ & _ \\ _ & _ & _ \end{bmatrix}$$

Solution:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

(c) **Design a 3x3 filter to blur an image.**

(Hint: blurring involves averaging a pixel's value with those of its neighbors.)

$$\begin{bmatrix} _ & _ & _ \\ _ & _ & _ \\ _ & _ & _ \end{bmatrix}$$

Solution:

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

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

(d) In a regular convolutional operation, the kernel slides over the input data in a contiguous manner. However, in dilated convolution, the kernel is "dilated" by introducing gaps between its elements, resulting in a larger receptive field for each output pixel.

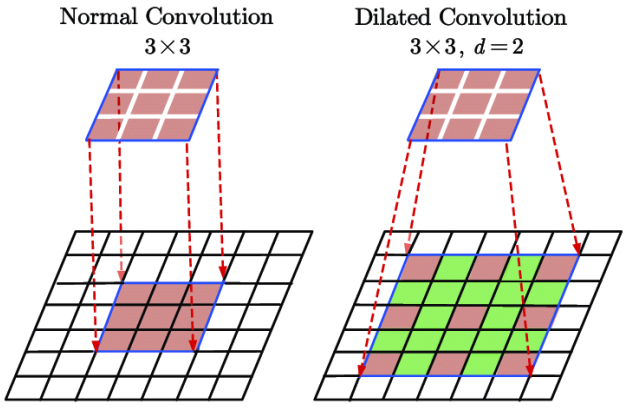


Figure 1: Dilated convolution kernels. Source: "Brain MRI Super-Resolution Using 3D Dilated Convolutional Encoder–Decoder Network" by Du et al.

The dilation (d) determines the spacing between kernel elements — a dilation of d introduces $d-1$ gaps between two kernel elements. The example above illustrates a 3x3 1-dilated kernel ($d=1$ is equivalent to a regular convolutional kernel) and a 3x3 2-dilated ($d=2$) kernel.

i. You are given an input matrix M and 2x2 filter k below. **Compute their dilated convolution with $d=2$.** Assume that gaps in the kernel are filled with zeros, stride is 1 and padding is 0.

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad k = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Solution: With dilation $d=2$, the kernel becomes a 3x3 kernel:

$$k_d = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Applying regular convolution $M * k_d$, we get:

$$M * k_d = \begin{bmatrix} 20 \end{bmatrix}$$

ii. Consider a two-layer architecture:

`DilatedConv1(3x3, d=1)` → `DilatedConv2(3x3, d=2)`

Both layers use the same sized kernel (3x3), but `DilatedConv1` has a dilation $d=1$ and `DilatedConv2` has a dilation $d=2$. **Compute the size of the receptive field at the output of the final layer (DilatedConv2).**

Recall that the receptive field is the region in the *original input* image whose pixels affect the output for a pixel in the specified layer.

Solution: 7x7. Drawing the outputs might help.

8. Homework Process and Study Group

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

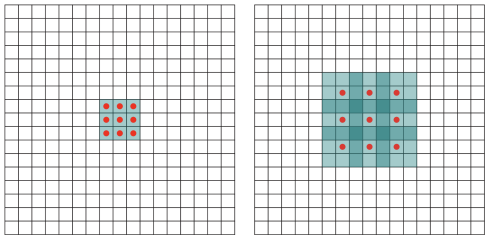


Figure 2: Receptive field size at DilatedConv1 and DilatedConv2 layers. Source: "Multi-Scale Context Aggregation by Dilated Convolutions", Yu et al.

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: Gireeja Ranade, Sizhe Chen, Matteo Guarrera, Mert Cemri, Kevin Frans, Anant Sahai, Suhong Moon, Dominic Carrano, Babak Ayazifar, Sukrit Arora, Romil Bhardwaj, Fei-Fei Li, Sheng Shen, Jake Austin, Kevin Li.

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