EECS 182 Deep Neural Networks
Fall 2026 Gireeja Ranade

Homework 6

This homework is due on Oct 14, at 10:59PM.

1. 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) In this experiment, we initialize the two convolutional layers with edge detectors and freeze both convolutional layers, so only the final layer (classifier) is trained. In the high data regime, this CNN performs worse than the CNN from the previous experiment, which was initialized with random weights and trained end to end. 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 can increase 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 can be 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 can dominate 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 95% for 30 images per class.

Solution: There is no definite answer in this question. If you achieve 95% for 30 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 while validation accuracy increasing. How can we interpret this?

Solution: As the model overfits to training set, the model can become overconfident on its remaining mistakes.

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

(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] as the edge width w and [$0 \sim 28-w$] as the edge location for each edge. Hence there are $28 + 27 + 26 + 25 = 106$ possible cases for each edge orientation. CNN achieves almost 100% validation accuracy with about 100 training images per class, 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 validation 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 pixel locations of the edges it saw during training, which all lie in the left or upper half 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 uses shared convolutional filters and global average pooling. 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 may fail 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 disrupted by the random permutation. Therefore, CNN may fail to learn the task. However, MLP is able to represent the same function after a corresponding permutation of its input weights because it is 'fully connected'. Therefore, MLP can learn the task.

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

Solution: The validation accuracy may decrease. This is because as kernel gets smaller, convolutional layer becomes less 'fully connected'. Therefore, CNN can have a harder time learning the task when the local correlation is disrupted by the random permutation.

(iii) What 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 may struggle to capture the local correlation of the image. Recalling backpropagation of CNN, CNN kernels are optimized to reduce the classification loss. (Any reasonable answer is also correct)

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

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 per class, max pooling performs better than average pooling. With 100 training images per class, average pooling is on par with max pooling.
Average pooling method smooths out the feature map and hence the sharp features may not be identified when this pooling method is used. Max pooling selects the strongest activation in each spatial window of each feature map. It is useful when we want to preserve a strong feature response within each pooling window. 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.

2. Weights and Gradients in a CNN

In this homework assignment, we aim to accomplish two objectives. Firstly, we seek to comprehend that the weight updates of a convolutional layer are a weighted sum of the input patches 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 $\mathbf{X} \in \mathbb{R}^{n \times n}$,

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

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

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

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

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,m} \end{bmatrix}. \tag{3}$$

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

For simplicity, we assume the number of the input channel (of $\mathbf{X}$) 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,

$$y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}, \tag{4}$$

or

$$\mathbf{Y} = \mathbf{X} * \mathbf{w}, \tag{5}$$

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 $d\mathbf{Y} \in \mathbb{R}^{m \times m}$,

$$d\mathbf{Y} = \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix}, \tag{6}$$

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

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

$$d\mathbf{w} = \begin{bmatrix} dw_{1,1} & dw_{1,2} & \cdots & dw_{1,k} \\ dw_{2,1} & dw_{2,2} & \cdots & dw_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ dw_{k,1} & dw_{k,2} & \cdots & dw_{k,k} \end{bmatrix}, \tag{7}$$

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

$$y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}. \tag{8}$$

Use chain rule

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

$$\frac{\partial \mathcal{L}}{\partial w_{h,l}} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{h,l}}, \tag{9}$$

and

$$\frac{\partial y_{i,j}}{\partial w_{h,l}} = x_{i+h-1,j+l-1}, \tag{10}$$

so we have

$$dw_{h,l} = \sum_{i=1}^{m} \sum_{j=1}^{m} x_{i+h-1,j+l-1} dy_{i,j} \tag{11}$$

$$d\mathbf{w} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix} * \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix} \tag{12}$$

$$= \mathbf{X} * d\mathbf{Y}. \tag{13}$$

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

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta d\mathbf{w} = \mathbf{w_t} - \eta \mathbf{X} * d\mathbf{Y} \tag{14}$$

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

$$\mathbb{E}[x_{i,j}] = 0, \tag{15}$$

$$\mathrm{Var}(x_{i,j}) = \sigma_x^2, \tag{16}$$

$$\mathbb{E}[dy_{i,j}] = 0, \tag{17}$$

$$\mathrm{Var}(dy_{i,j}) = \sigma_g^2. \tag{18}$$

Derive the mean and variance of $dw_{i,j} = \dfrac{\partial \mathcal{L}}{\partial w_{i,j}}$ for each i, j as a function of n, k, σ_x, σ_g. What is the asymptotic growth rate of the standard deviation of the gradient on $dw_{i,j}$ with respect to the length and width of the image n, with k, $\sigma_x > 0$, and $\sigma_g > 0$ fixed?

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.

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

Solution:

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

$$dw_{i,j} = \sum_{h=1}^{m}\sum_{l=1}^{m} dy_{h,l}x_{i+h-1,j+l-1}. \tag{19}$$

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:

$$\begin{aligned}\mathbb{E}[dw_{i,j}] &= \sum_{h=1}^{m}\sum_{l=1}^{m} \mathbb{E}[dy_{h,l}]\mathbb{E}[x_{i+h-1,j+l-1}] &(20)\\ &= 0. &(21)\end{aligned}$$

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

$$\begin{aligned}\mathrm{Var}(dw_{i,j}) &= \mathbb{E}[dw_{i,j}^2] - \mathbb{E}[dw_{i,j}]^2 &(22)\\ &= \mathbb{E}[dw_{i,j}^2] - 0 &(23)\\ &= \mathbb{E}\left[\left(\sum_{h=1}^{m}\sum_{l=1}^{m} dy_{h,l}x_{i+h-1,j+l-1}\right)^2\right] &(24)\\ &= \sum_{h=1}^{m}\sum_{l=1}^{m} \mathbb{E}[dy_{h,l}^2]\mathbb{E}[x_{i+h-1,j+l-1}^2] &(25)\\ &= \sum_{h=1}^{m}\sum_{l=1}^{m} (\mathrm{Var}(dy_{h,l}) + \mathbb{E}[dy_{h,l}]^2)(\mathrm{Var}(x_{i+h-1,j+l-1}) + \mathbb{E}[x_{i+h-1,j+l-1}]^2) &(26)\\ &= \sum_{h=1}^{m}\sum_{l=1}^{m} \mathrm{Var}(dy_{h,l})\mathrm{Var}(x_{i+h-1,j+l-1}) &(27)\\ &= m^2\sigma_x^2\sigma_g^2. &(28)\end{aligned}$$

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

$$\mathrm{Std}(dw_{i,j}) = \sqrt{\mathrm{Var}(dw_{i,j})} = \sqrt{(n-k+1)^2\sigma_x^2\sigma_g^2} = \Theta(n), \tag{29}$$

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

(c) Assume stride 2, no padding, and an even input side length. **For a network with only one 2x2 max-pooling layer (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 one 2x2 average-pooling layer (no convolution layers, no activations), what will be $d\mathbf{X}$?**

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

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.*

$$y_{1,1} = x_{1,1} = \max(x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}) \tag{30}$$

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,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } (i,j) = (1,1) \\ 0, & \text{otherwise} \end{cases} \tag{31}$$

Combining all four partial derivatives, we have

$$d\mathbf{X} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} dy_{11} & 0 \\ 0 & 0 \end{bmatrix} \tag{32}$$

For the average pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = 1/4 \tag{33}$$

$$d\mathbf{X} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \\ \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \end{bmatrix} \tag{34}$$

In the general setting, for max pooling, we can notice that x and y have a many to one mapping. Each x value is involved in calculation of exactly one y value. Let $k = \lceil i/2 \rceil$, $l = \lceil j/2 \rceil$, so that $x_{i,j}$ lies in the 2×2 block that produces y_{kl}, and assume the maximum in each block is unique. Let

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } x_{i,j} = \max(x_{2k-1,2l-1}, x_{2k-1,2l}, x_{2k,2l-1}, x_{2k,2l}) \\ 0, & \text{otherwise} \end{cases} \tag{35}$$

, then

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} \tag{36}$$

$d\mathbf{X}$ is the matrix constructed 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 dy_{kl} flowing back, and the rest three inputs have zero gradient.

For the average pooling general case,

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \frac{1}{4} \tag{37}$$

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

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} = \frac{dy_{kl}}{4} \tag{38}$$

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

(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 gives some robustness to small shifts of a feature within the pooling window. 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 can help capture smooth transitions or subtle changes in the feature map. For example, average pooling usually follows the last convolution layer of feature learning.

3. Convolutional Networks

Note: Throughout this problem, we will use the convention of NOT flipping the filter before dragging it over the signal. This is the standard notation with neural networks (technically, this operation is cross-correlation)

(a) **List two reasons we typically prefer convolutional layers instead of fully connected layers when working with image data.**

Solution: The basic reason is so that we can get an inductive bias that is suited for the regularities often present in image data so that we can learn without having to use inordinate amounts of training data. Valid reasons for using convolutional layers to do this include:

- Convolutions with finite sized filters respect the kind of locality and by layering them, we can reflect the hierarchical part/whole structure decompositions that we see in images.
- Convolutions are nicely compatible with weight-sharing which builds in a basic equivariance to translations which is reasonable since in most image-related contexts, the desired answers for a translated image are closely related to the original image. The weight-sharing allows the numbers of parameters to be much smaller.
- Convolutions build in the idea that left/right and up/down can be different and allow different weights for those.
- There is a long history of using filterbanks and convolutional structures in hand-designed approaches to computer vision and image processing so it is natural to simply replace those with learned weights in similar structures.
- There is biological inspiration based on what we know about how the visual system works for the locality properties and the kinds of edge-detecting responses that convolutions capture naturally.
- Furthermore, other people have used convolutional layers to great success when working with vision and in Deep Learning, it is natural to try to build on the successes of others even when we don't understand why they succeeded.

(b) Consider the following 1D signal: $[1, 4, 0, -2, 3]$. After convolution with a length-3 filter, no padding, stride=1, we get the following sequence: $[-2, 2, 11]$. **What was the filter?**

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

(Hint: Just to help you check your work, the first entry in the filter that you should find is 2. However, if you try to use this hint directly to solve for the answer, you will not get credit since this hint only exists to help you check your work.)

Solution: If we represent the original filter as $[h_1, h_2, h_3]$, we can set up the system of equations:

$$\begin{aligned} h_1 + 4h_2 + 0h_3 &= -2 \\ 4h_1 + 0h_2 - 2h_3 &= 2 \\ 0h_1 - 2h_2 + 3h_3 &= 11 \end{aligned}$$

Solving, we get $[h_1, h_2, h_3] = [2, -1, 3]$

This could be done using Gaussian elimination or substitution. By substitution, we see $h_1 = -2 - 4h_2$ and $h_3 = \frac{11}{3} + \frac{2}{3}h_2$. Plugging into the second equation we get $-8 - 16h_2 - \frac{22}{3} - \frac{4}{3}h_2 = 2$ which simplifies to $\frac{52}{3} = -\frac{52}{3}h_2$ which solves to $h_2 = -1$. Back substituting gives us $h_1 = 2$ and $h_3 = \frac{9}{3} = 3$.

The Gaussian elimination approach involves similar numbers.

(c) Transpose convolution is an operation to help us upsample a signal (increase the resolution). For example, if our original signal were $[a, b, c]$ and we perform transpose convolution with pad=0 and stride=2, with the filter $[x, y, z]$, the output would be $[ax, ay, az + bx, by, bz + cx, cy, cz]$. Notice that the entries of the input are multiplied by each of the entries of the filter. Overlaps are summed. Also notice how for a fixed filter size and stride, the dimensions of the input and output are swapped compared to standard convolution. (For example, if we did standard convolution on a length-7 sequence with filter size of 3 and stride=2, we would output a length-3 sequence).

If our 2D input is $\begin{bmatrix} -1 & 2 \\ 3 & 1 \end{bmatrix}$ and the 2D filter is $\begin{bmatrix} +1 & -1 \\ 0 & +1 \end{bmatrix}$ **What is the output of transpose convolution with pad=0 and stride=1?**

Solution: The output is $\begin{bmatrix} -1 & 3 & -2 \\ 3 & -3 & 1 \\ 0 & 3 & 1 \end{bmatrix}$.

The easiest way to think about transpose convolution is that every input entry stamps a copy of the filter onto the output, scaled by its own value. The stride says how far apart the stamps start, and where stamps overlap we add them up. Padding works the other way around from ordinary convolution: pad $= p$ trims p entries off each edge of the result, so pad $= 0$ means we keep all of it.

In the 1D example from the question, the entry a stamps ax, ay, az onto positions $0, 1, 2$. Because the stride is 2, the stamp for b starts two positions later, so bx, by, bz land on positions $2, 3, 4$, and cx, cy, cz land on $4, 5, 6$. Positions 2 and 4 each get two contributions, which is where $az + bx$ and $bz + cx$ come from, and the output has 7 entries in total.

This is also why transpose convolution is used for upsampling. With stride 2, each new input entry moves the stamp two positions further along, so the output is roughly twice as long as the input, and the filter decides how to fill in the new positions. For example, a filter as long as the stride, $[1, 1]$ with stride 2, turns $[a, b, c]$ into $[a, a, b, b, c, c]$, which just repeats every value. In a network the filter is learned, so the network learns its own way of filling in.

You will also see transpose convolution described as "insert zeros, then convolve." In the example, the stamp of a starts at position 0, the stamp of b at position 2, and the stamp of c at position 4. Write each input at the position where its stamp starts and put 0 in the positions in between: that gives $[a, 0, b, 0, c]$. With stride s the stamps start s positions apart, so there are always $s - 1$ zeros between neighboring inputs. Next, for a filter of length k, add $k - 1 = 2$ zeros at each end to get $[0, 0, a, 0, b, 0, c, 0, 0]$ (these

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

are part of the trick, not the pad setting). Without them, the first window would already include b; with them, the first window $[0, 0, a]$ touches only a and the last window $[c, 0, 0]$ touches only c. Finally, slide the reversed filter $[z, y, x]$ across this sequence one position at a time. The windows $[0, 0, a]$, $[0, a, 0]$ and $[a, 0, b]$ give ax, ay and $az + bx$, and so on: the same seven numbers as before. The filter is reversed because, as the window slides to the right, each input enters at the right end of the window and leaves at the left end, so it meets x first, then y, then z, just like its stamp.

For the 2D problem, the rule is the same as in 1D, just applied to both rows and columns: the input entry in row i, column j (counting from 0) stamps its scaled copy of the filter with the top-left corner at row $s \cdot i$, column $s \cdot j$ of the output. With stride 1 that is just row i, column j. The stamps are 2×2 and their corners are at rows 0 or 1 and columns 0 or 1, so together they cover a 3×3 output. With pad $= 0$ we keep all of it. Here are the four stamps, one input entry at a time:

-1 (row 0, column 0):
$$(-1)\begin{bmatrix}1 & -1\\ 0 & 1\end{bmatrix} = \begin{bmatrix}-1 & 1\\ 0 & -1\end{bmatrix} \longrightarrow \begin{bmatrix}-1 & 1 & 0\\ 0 & -1 & 0\\ 0 & 0 & 0\end{bmatrix}$$

2 (row 0, column 1):
$$2\begin{bmatrix}1 & -1\\ 0 & 1\end{bmatrix} = \begin{bmatrix}2 & -2\\ 0 & 2\end{bmatrix} \longrightarrow \begin{bmatrix}0 & 2 & -2\\ 0 & 0 & 2\\ 0 & 0 & 0\end{bmatrix}$$

3 (row 1, column 0):
$$3\begin{bmatrix}1 & -1\\ 0 & 1\end{bmatrix} = \begin{bmatrix}3 & -3\\ 0 & 3\end{bmatrix} \longrightarrow \begin{bmatrix}0 & 0 & 0\\ 3 & -3 & 0\\ 0 & 3 & 0\end{bmatrix}$$

1 (row 1, column 1):
$$1\begin{bmatrix}1 & -1\\ 0 & 1\end{bmatrix} = \begin{bmatrix}1 & -1\\ 0 & 1\end{bmatrix} \longrightarrow \begin{bmatrix}0 & 0 & 0\\ 0 & 1 & -1\\ 0 & 0 & 1\end{bmatrix}$$

Each stamp sits in its own corner of the grid, with zeros wherever it doesn't reach. Adding the four grids entry by entry gives the answer,

$$\begin{bmatrix}-1 & 3 & -2\\ 3 & -3 & 1\\ 0 & 3 & 1\end{bmatrix}.$$

For example, the center entry gets something from every stamp: $-1 + 0 - 3 + 1 = -3$.

To see the upsampling in 2D, try the same input and filter with stride 2. Now the corners are at rows 0 or 2 and columns 0 or 2, so the stamps don't overlap: each input entry becomes its own 2×2 block and the 2×2 input turns into a 4×4 output:

$$\begin{bmatrix}-1 & 1 & 2 & -2\\ 0 & -1 & 0 & 2\\ 3 & -3 & 1 & -1\\ 0 & 3 & 0 & 1\end{bmatrix}.$$

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

4. Memory considerations when using GPUs (Coding Question)

In this homework, you will run GPUMemory.ipynb to train a ResNet model on CIFAR-10 using PyTorch and explore its implications on GPU memory.

We will explore various systems considerations, such as the effect of batch size on memory usage and how different optimizers (SGD, SGD with momentum, Adam) vary in their memory requirements.

It is strongly recommended that you start early on this question, since colab daily GPU limits may require you to complete this question over a few days with breaks in between.

(a) Managing GPU memory for training neural networks (Notebook Section 1).

(i) **How many trainable parameters does ResNet-152 have**? What is the **estimated size of the model in MB**?

Solution: For the notebook's 10-class classifier, ResNet-152 has 58164298 parameters, and for float32 parameters, it is estimated to be 232MB in size.

(ii) Which GPU are you using? **How much total memory does it have**?

Solution: Colab may provide a Nvidia T4 or another GPU; report the device and capacity shown by your runtime.

(iii) After you load the model into memory, **what is the memory overhead (MB) beyond the model parameters**?

Solution: Report the measured overhead. This includes model buffers and allocator/library allocations, not just the CUDA context, and varies by runtime.

(b) Optimizer memory usage (Notebook Section 2).

(i) **What is the total memory utilization during training with SGD, SGD with momentum and Adam optimizers**? Report in MB individually for each optimizer.

Solution: Generally SGD would have lowest memory utilization, followed by SGD with momentum. Adam would typically have maximum memory usage. Example values from a Colab T4 GPU (1 MiB = 1.048576 MB): SGD: 3192.0 MiB, SGD with momentum: 3274.0 MiB, ADAM: 3358.0 MiB.

(ii) **Which optimizer consumes the most memory**? Why?

Solution: ADAM consumes more memory than SGD or SGD with momentum because it stores two moving averages of the gradients and their squares, which are used to estimate the first and second moments of the gradient. Because these moving averages are updated in each step, and they need to be stored in memory.

(c) Batch size, learning rates and memory utilization (Notebook Section 3)

(i) **What is the memory utilization for different batch sizes (4, 16, 64, 256)? What is the largest batch size you were able to train?**

Solution: Exact answers depend on the GPU used by the student. Example memory utilizations (trends are important, not absolute values): 4: 1406.0 MiB, 16: 1706.0 MiB, 64: 2738.0 MiB, 256: 8744.0 MiB. Batch sizes >= 1024 may result in an out of memory error on a T4 GPU.

(ii) **Which batch size gave you the highest accuracy at the end of 10 epochs?**

Solution: Answers may vary, depending on the measured results.

(iii) **Which batch size completed 10 epochs the fastest (least wall clock time)? Why?**

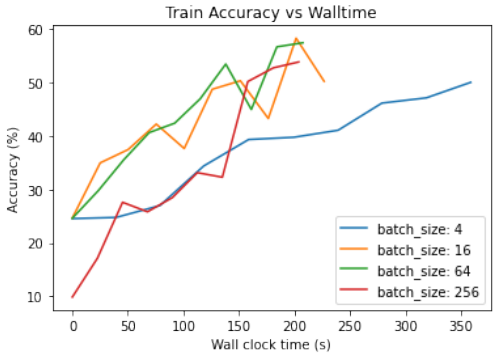
Solution: The largest batch size run (256 or 512) may have completed 10 epochs fastest because they can take advantage of hardware acceleration and parallelism, and they can use memory

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

more efficiently. Specifically, GPUs are based on the SIMD (Single Instruction, Multiple Data) paradigm. This means that if we have a large batch of data, we can perform the same operation on each piece of data in parallel, increasing the throughput we see.

(iv) Attach your **training accuracy vs wall time plots** with your written submission.

Solution: Exact plots may vary. Example accuracy vs wall time plot:



5. Denoising using U-Nets

U-Nets are one of the most influential architectures in deep learning for tasks requiring precise spatial localization—such as image segmentation and denoising. The U-Net is composed of two main components:

Encoder: aims to decrease the spatial dimensions of the image into a feature map that captures relevant information about the image.

Decoder: aims to upsample the feature map and produce a relevant segmentation map using the patterns learnt in the contracting path.

Importantly, U-Nets add **skip connections** such that every block in the decoder incorporates the feature map from its corresponding block in the encoder. Its architecture is visualized in the below figure:

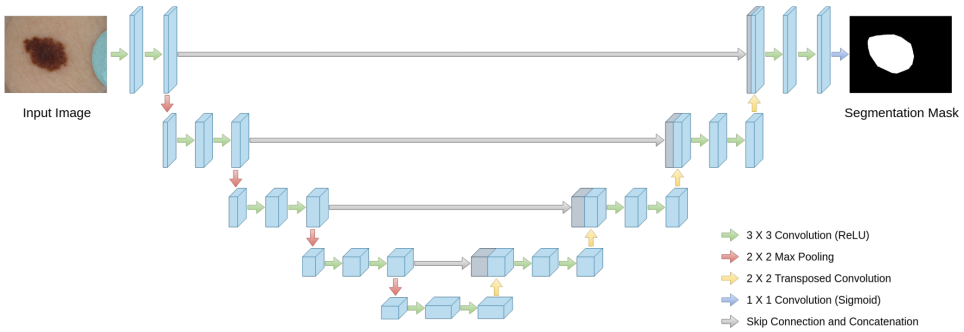


Figure 1: Diagram for U-Net

To understand why U-Nets are so effective, it helps to study their components in a simplified setting of denoising a 1D signal.

Let a clean signal be $\mathbf{x} = [0, 0, 0, 9, 9, 9, 9, 9]^T$. As input, we observe $\hat{\mathbf{x}} = \mathbf{x} + \boldsymbol{\epsilon}$, where each ϵ_i is independent Gaussian noise with variance $\mathrm{Var}(\epsilon_i) = \sigma^2$. Our goal is to predict $\mathbf{y}$ such that it is as close to the true signal $\mathbf{x}$ as possible.

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

(a) Consider a simple U-net architecture with 2 levels where each encoder block performs a convolution with kernel $\mathbf{k} = \left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]^T$ using stride 1 and same padding. In between the two blocks is max-pooling with filter size 2 and stride 2. Let $\mathbf{e}$ be the output of the encoder applied to $\hat{\mathbf{x}}$. **What is the expected output $\mathbb{E}[\mathbf{e}]$? What is the variance of each element e_i (in terms of σ)?** You can assume that elements of the input to each convolution are independently and identically distributed.

Solution: [Correction: the complex correct solution that does not teach what we intend to teach]
First convolution and max-pooling. Write $h_i = (\hat{x}_{i-1} + \hat{x}_i + \hat{x}_{i+1})/3$, with $\hat{x}_0 = \hat{x}_9 = 0$. Then

$$\mu = \mathbb{E}[h] = [0, 0, 3, 6, 9, 9, 9, 6]^T, \qquad \mathrm{Var}(h_i) = \frac{\sigma^2}{9} b_i, \quad b = [2, 3, 3, 3, 3, 3, 3, 2]^T.$$

For each pooling pair define

$$S_j = h_{2j-1} + h_{2j}, \qquad D_j = h_{2j-1} - h_{2j}, \qquad p_j = \max(h_{2j-1}, h_{2j}) = \frac{S_j + |D_j|}{2}.$$

The four differences are

$$D_1 = -\frac{\epsilon_3}{3}, \quad D_2 = -3 + \frac{\epsilon_2 - \epsilon_5}{3}, \quad D_3 = \frac{\epsilon_4 - \epsilon_7}{3}, \quad D_4 = 3 + \frac{\epsilon_6}{3}.$$

They are independent because they involve disjoint input-noise variables. Their means and standard deviations are

$$d = [0, -3, 0, 3]^T, \qquad \tau = \frac{\sigma}{3}[1, \sqrt{2}, \sqrt{2}, 1]^T.$$

Let $\phi(t) = (2\pi)^{-1/2} e^{-t^2/2}$ and $\Phi(t) = \int_{-\infty}^{t} \phi(u)\, du$. For $\sigma > 0$, define

$$g_j = \mathbb{E}|D_j| = 2\tau_j \phi(d_j/\tau_j) + d_j\big(2\Phi(d_j/\tau_j) - 1\big),$$
$$v_j = \mathrm{Var}(|D_j|) = d_j^2 + \tau_j^2 - g_j^2.$$

Thus the pooled mean is

$$m = \mathbb{E}[p] = \frac{1}{2}[g_1,\ 9 + g_2,\ 18 + g_3,\ 15 + g_4]^T.$$

For example, $m_1 = \sigma/(3\sqrt{2\pi}) > 0$, although the first pooled value of the clean signal is zero. This directly exhibits the max-pooling bias.

Expected encoder output. The second convolution is linear. With

$$B = \frac{1}{3}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \qquad e = Bp,$$

the expected encoder output is

$$\boxed{\mathbb{E}[e] = Bm = \frac{1}{3}[m_1 + m_2,\ m_1 + m_2 + m_3,\ m_2 + m_3 + m_4,\ m_3 + m_4]^T.}$$

This expectation depends on σ. In particular, $m_2 \geq 6$ and $m_1 > 0$, so $\mathbb{E}[e_1] > 2$ whenever $\sigma > 0$.

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

Encoder variances. Let $w_j = \mathrm{Var}(p_j)$ and $r = 2\Phi(9/\sigma) - 1$. The pooled marginal variances are

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \frac{\sigma^2}{36} \begin{bmatrix} 9 \\ 10 \\ 10 \\ 9 + 2r \end{bmatrix} + \frac{1}{4} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}.$$

Indeed, the sums S_j have variances $\sigma^2[9, 10, 10, 9]^T/9$, and

$$\mathrm{Cov}(S_j, |D_j|) = \mathrm{Cov}(S_j, D_j)\big(2\Phi(d_j/\tau_j) - 1\big).$$

Here $\mathrm{Cov}(S_j, D_j) = \sigma^2 c_j/9$, where $c = [-1, 0, 0, 1]^T$, so only the fourth pair contributes a nonzero sum–absolute-difference covariance. Applying

$$w_j = \frac{1}{4}\left(\mathrm{Var}(S_j) + v_j + 2\,\mathrm{Cov}(S_j, |D_j|)\right)$$

gives the displayed variances.

Using the independence assumption in part (a) for the inputs to the second convolution, the four encoder variances $V_j = \mathrm{Var}(e_j)$ are

$$\boxed{\begin{aligned} V_1 &= (w_1 + w_2)/9, \\ V_2 &= (w_1 + w_2 + w_3)/9, \\ V_3 &= (w_2 + w_3 + w_4)/9, \\ V_4 &= (w_3 + w_4)/9. \end{aligned}}$$

These are functions of σ through g_j, v_j, r. The first and last outputs average two random terms and one padded zero. When $\sigma = 0$, use $m = [0, 6, 9, 9]^T$, $\mathbb{E}[e] = [2, 5, 8, 6]^T$, and $V_j = 0$.

(b) In the same U-net, the decoder performs a transpose convolution with kernel $\mathbf{k} = [1, 1]^T$ with stride 2. Then, the reconstructed signal is the element-wise average of the upsampled signal and the output of the first encoder block (via a skip connection). **What is the expected reconstructed signal $\mathbb{E}[\mathbf{y}]$? What is the variance of each element y_i Why is the skip connection important?** You can assume that the upsampled signal and the output of the encoder block are independent.

(Hint: The smoothing operation in the encoder, though useful for dampening noise, destroys one crucial detail about the signal. The skip connection is important to bring this detail.)

Solution: Let $a = \mathbb{E}[e] = Bm$ from part (a) and $k(i) = \lceil i/2 \rceil$. Transpose convolution repeats each encoder value, so $u_i = e_{k(i)}$ and $y_i = (u_i + h_i)/2$. Therefore

$$\boxed{\mathbb{E}[y] = \frac{1}{2}[a_1,\ a_1,\ a_2 + 3,\ a_2 + 6,\ a_3 + 9,\ a_3 + 9,\ a_4 + 9,\ a_4 + 6]^T.}$$

At $\sigma = 0$, this is $[1, 1, 4, 5.5, 8.5, 8.5, 7.5, 6]^T$.

Using the independence assumption in part (b),

$$\boxed{\mathrm{Var}(y_i) = \frac{1}{4}\left(V_{k(i)} + \frac{\sigma^2 b_i}{9}\right).}$$

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

All these variances are zero in the noiseless case.

What the skip connection contributes. The upsampled branch has $u_3 = u_4$, so it cannot distinguish these two locations. The skip branch retains their difference: $y_4 - y_3 = (h_4 - h_3)/2$ and hence $\mathbb{E}[y_4 - y_3] = 3/2$. It supplies some information about the transition, but does not recover the clean jump of 9 exactly. It also transmits noise. A skip connection can preserve finer spatial information; fixed averaging does not guarantee lower reconstruction error or make both branches necessary for every signal.

Contributors: Matteo Guarrera, Mert Cemri, Sizhe Chen, Suhong Moon, Kumar Krishna Agrawal, Peter Wang, Anant Sahai, Qiyang Li, Linyuan Gong, Long He, Saagar Sanghavi, Romil Bhardwaj, Joey Hong, Gireeja Ranade.

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