HW4.

Mengying Lin SID: 3038737132.

1. Understanding Dropout (Coding Question)

In this question, you will analyze the effect of dropout in a simplified setting. Please follow the instructions in the Jupyter notebook and answer the questions in your submission of the written assignment. The notebook does not need to be submitted.

(a) (No dropout, least-square) **The mathematical expression of the OLS solution, and the solution calculated in the code cell.**

$$w = (X^T X)^{-1} X^T y.$$

$$\text{Result} = \begin{pmatrix} 1.08910891 \\ 0.10891089 \end{pmatrix}.$$

(b) (No dropout, gradient descent) **The solution in the code cell. Are the weights obtained by training with gradient descent the same as those calculated using the closed-form least squares method**

net = nn.Linear(2,1).

The result is $\begin{pmatrix} 1.0784 \\ 0.1078 \end{pmatrix}$, which is quite close to the weights obtained in (a).

(c) (Dropout, least-square) **The solution in the code cell.**

```
[19] ###############################################################################
     #  YOUR  CODE  HERE
     ###############################################################################
     x  =  np.array([[0,  1],  [10,  0],  [0,  0],  [10,  1]])  *  2
     y  =  np.array([[11],  [11],  [11],  [11]])
     w  =  np.linalg.pinv(x.T  @  x)  @  x.T  @  y
     ###############################################################################
     print("x  =",  x)
     print("y  =",  y)
     print("w  =",  w)
```

```
x = [[ 0  2]
 [20  0]
 [ 0  0]
 [20  2]]
y = [[11]
 [11]
 [11]
 [11]]
w = [[0.36666667]
 [3.66666667]]
```

(d) (Dropout, gradient descent) **Describe the shape of the training curve. Are the weights obtained by training with gradient descent the same as those calculated using the closed-form least squares method?**

The training curve fluctuates dramatically, and does not converge.

The weights obtained by GD is quite different from those of the closed form least squares method.

(e) (Dropout, gradient descent, large batch size) **Describe the loss curve and compare it with the loss curve in the last part. Why are they different? Also compare the trained weights with the one calculated by the least-square formula.**

The loss curve goes downwards rapidly at the first stage, then in a relatively slow pace, and finally slightly fluctuates.

The batchsize is larger, so the true distribution is much closer to the estimation.

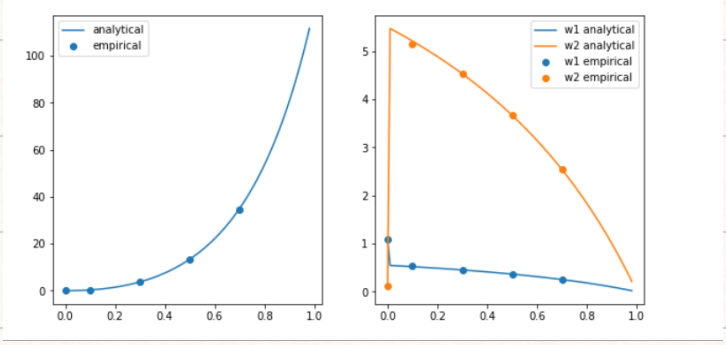
(f) Refer back to the cells you ran in part (e). **Analyze how and why adding dropout changes the following: (i) How large were the final weights w_1 and w_2 compared to each other. (ii) How large the contribution of each term (i.e. $10w_1$ + w_2) is to the final output. Why does this change occur?** (This does not need to be a formal math proof).

i> If we randomly drop out some components of the input, it is like we "freeze" the weights temporarily, and during backpropgation, the gradients are 0, and the weights won't be updated. At the same time, the remained one's will be rescaled. Therefore it might change the final weights.

ii). For the reasons mentioned above, it might also change

the contribution of each term.

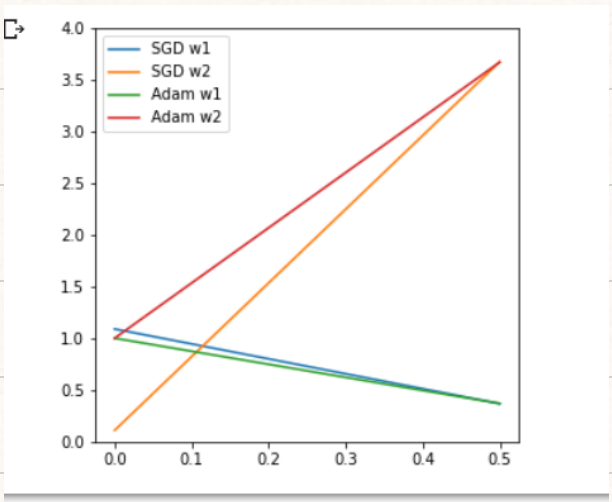
(g) (Optional) Sweeping over the dropout rate **Fill out notebook section (G).** You should see that as the dropout rate changes, w_1 and w_2 change smoothly, except for a discontinuity when dropout rates are 0. Explain this discontinuity.



$w = (X^T S X)^{-1} X^T S y$, $S = \begin{pmatrix} p(1-p) & & \\ & p(1-p) & \\ & & p^2 \\ & & & (1-p)^2 \end{pmatrix}$

Note when $p=0$, $X^T S X$ is not full rank, we need to use pseudoinverse. (Not sure ...)

(h) (Optional) Optimizing with Adam: Run the cells in part (H). **Does the solution change when you switch from SGD to Adam? Why or why not?**



The solution does not change when p set to 0.5, since Adam only adjusts lr and won't affect the loss, which is related with the gradients.

[Q: why the result is different when p = 0?]

(i) Dropout on real data: **Run the notebook cells in part (I), and report on how they affect the final performance.**

For model without dropout, the accuracy on data with cheating features are extremely high (100%), while it is way more lower on clean data (10%)

For model with dropout, the acc on cheating data is

93%, and that on clean data is >1%.

2. Regularization and Dropout

You saw one perspective on the implicit regularization of dropout in HW, and here, you will see another one. Recall that linear regression optimizes the following learning objective:

$$\mathcal{L}(\mathbf{w}) = ||\mathbf{y} - X\mathbf{w}||_2^2 \tag{1}$$

One way of using *dropout* during SGD on the d-dimensional input features $\mathbf{x}_i$ involves *keeping* each feature at random $\sim_{i.i.d}$ $Bernoulli(p)$ (and zeroing it out if not kept) and then performing a traditional SGD step.

It turns out that such dropout makes our learning objective effectively become

$$\mathcal{L}(\check{\mathbf{w}}) = \mathbb{E}_{R\sim Bernoulli(p)}\left[||\mathbf{y} - (R \odot X)\check{\mathbf{w}}||_2^2\right] \tag{2}$$

where $\odot$ is the element-wise product and the random binary matrix $R \in \{0,1\}^{n\times d}$ is such that $R_{i,j} \sim_{i.i.d}$ $Bernoulli(p)$. We use $\check{\mathbf{w}}$ to remind you that this is learned by dropout.

Recalling how Tikhonov-regularized (generalized ridge-regression) least-squares problems involve solving:

$$\mathcal{L}(\mathbf{w}) = ||\mathbf{y} - X\mathbf{w}||_2^2 + ||\Gamma\mathbf{w}||_2^2 \tag{3}$$

for some suitable matrix Γ.

(a) **Show that we can manipulate (2) to eliminate the expectations and get:**

$$\mathcal{L}(\check{\mathbf{w}}) = ||\mathbf{y} - pX\check{\mathbf{w}}||_2^2 + p(1-p)||\check{\Gamma}\check{\mathbf{w}}||_2^2 \tag{4}$$

with $\check{\Gamma}$ being a diagonal matrix whose j-th diagonal entry is the norm of the j-th column of the training matrix X.

as Denote $R\odot X$ by α

$$\mathcal{L}(\check{w}) = E[(y-\alpha\check{w})^T(y-\alpha\check{w})]$$

$$= E[y^Ty - \check{w}^T\alpha^Ty - y^T\alpha\check{w} + \check{w}^T\alpha^T\alpha\check{w}].$$

$$= y^Ty - \check{w}pX^Ty - y^TpX\check{w} + \check{w}^TE[\alpha^T\alpha]\check{w}.$$

$$E[\alpha^T\alpha]_{ij} = E[\textstyle\sum R_{ki}X_{ki}R_{kj}X_{kj}] = \sum_{k=1}^{n} E[R_{ki}R_{kj}]X_{ki}X_{kj}$$

$$= \begin{cases} \sum_{k=1}^{n} pX_{ki}^2 & (i=j) \\ \sum_{k=1}^{n} p^2X_{ki}X_{kj} & (i\neq j) \end{cases}$$

$$\mathcal{L}(\check{w}) = y^Ty - \check{w}pX^Ty - y^TpX\check{w} + \check{w}^TX^T\begin{bmatrix} p & \cdots & p^2 \\ p^2 & \ddots & p \end{bmatrix}X\check{w}$$

$$= y^Ty - \check{w}pX^Ty - y^TpX\check{w} + p^2\check{w}^TX^TX\check{w} + p(1-p)\check{w}^Tdiag(X^TX)\check{w}.$$

$= \|y - pX\check{w}\|_2^2 + p(1-p)\|\check{\Gamma}\check{w}\|_2^2.$

(b) **How should we transform the $\check{w}$ we learn using (4) (i.e. with dropout) to get something that looks a solution to the traditionally regularized problem (3)?**

(Hint: This is related to how we adjust weights learned using dropout training for using them at inference time. PyTorch by default does this adjustment during training itself, but here, we are doing dropout slightly differently with no adjustments during training.)

let $p\check{w} = w$. We expect $p(1-p)\|\check{\Gamma}\check{w}\|_2^2 = \|\Gamma w\|_2^2$

which means $\left\|\sqrt{\frac{p(1-p)}{p}}\check{\Gamma} w\right\|_2^2 = \|\Gamma w\|_2^2$.

Hence $\sqrt{\frac{1-p}{p}}\check{\Gamma} = \Gamma$.

(c) With the understanding that the Γ in (3) is an invertible matrix, **change variables in (3) to make the problem look like classical ridge regression:**

$$\mathcal{L}(\tilde{w}) = \|y - \tilde{X}\tilde{w}\|_2^2 + \lambda\|\tilde{w}\|_2^2 \tag{5}$$

Explicitly, what is the changed data matrix $\tilde{X}$ in terms of the original data matrix X and Γ?

$L(w) = \|y - Xw\|_2^2 + \|\Gamma w\|_2^2.$

Let $\|\Gamma w\|_2^2 = \|\sqrt{\Sigma}\hat{w}\|_2^2$, we have $\hat{w} = \frac{\Gamma w}{\sqrt{\Sigma}}$.

We expect $\|y - \hat{X}\hat{w}\|_2^2 = \|y - \hat{X}\frac{\Gamma w}{\sqrt{\Sigma}}\|_2^2 = \|y - Xw\|_2^2$.

Therefore $\frac{\hat{X}\Gamma}{\sqrt{\Sigma}} = X$,

$\hat{X} = \sqrt{\Sigma}\Gamma^{-1}X.$

(d) Continuing the previous part, with the further understanding that Γ is a *diagonal* invertible matrix with the j-th diagonal entry proportional to the norm of the j-th column in X, **what can you say about the norms of the columns of the effective training matrix $\tilde{X}$ and speculate briefly on the relationship between dropout and batch-normalization.**

$\hat{\Gamma} = \text{diag}(X^TX)$

$\Gamma = \sqrt{\frac{1-p}{p}}\hat{\Gamma} = \sqrt{\frac{1-p}{p}}\,\text{diag}(X^TX).$

$$\Gamma_{ii} = \sqrt{\frac{1-p}{p}} \sum_{k=1}^{n} x_{ki}^2.$$

$$\hat{X} = \sqrt{\lambda}\,\Gamma^{-1} X. \qquad \hat{x}_{ij} = \sqrt{\lambda}\sqrt{\frac{p}{1-p}} \left(\sum_{k=1}^{n} x_{ki}^2\right)^{-1} x_{ij}.$$

For colum $\hat{x}^j$: $\|\hat{x}^j\|_2 = \sqrt{\lambda}\sqrt{\frac{1}{p}-1}\sqrt{\sum_{i=1}^{n}\left[\left(\sum_{k=1}^{n} x_{ki}^2\right)^{-1} x_{ij}\right]^2}$

Intuitively, the larger p is, the larger $\|\hat{x}^j\|_2$ is.

Dropout will only rescale the input, but BN can also translate the data pts.

3. 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 $\mathbf{X} \in \mathbb{R}^{n \times n}$.

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

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

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

$dw_{ij} = dY \cdot \frac{\partial Y}{\partial w_{ij}}$.

$\frac{\partial y_{l,k}}{\partial w_{ij}} = x_{l-i+1,\, k-j+1}$.

Therefore $(dw_{ij})_{l,k} = dy_{l,k}\, x_{l-i+1,\, k-j+1}$

$w \leftarrow w - \eta\, dw$.

Here $b = 0$. $\frac{\partial y}{\partial b} = 0$. $b \leftarrow b$.

(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:

$$\mathbb{E}[x_{i,j}] = 0, \tag{13}$$
$$\mathrm{Var}(x_{i,j}) = \sigma_x^2, \tag{14}$$
$$\mathbb{E}[dy_{i,j}] = 0, \tag{15}$$
$$\mathrm{Var}(dy_{i,j}) = \sigma_g^2. \tag{16}$$

Derive the mean and variance of $dw_{i,j} = \dfrac{\partial \mathcal{L}}{\partial W_{i,j}}$ **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** $dw_{i,j}$ **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.*

Note $x_{i,j}$ and $dy_{i,j}$ are independent.

$E[(dw_{i,j})_{l,k}] = E[dy_{l,k}]\, E[x_{l-i+1,\,k-j+i}] = 0.$

Therefore $E[dw_{i,j}] = 0_{m\times n}$.

$Var[dw_{i,j}] = E[dw_{i,j} \cdot dw_{i,j}^T]$.

$$\left(Var[dw_{i,j}]\right)_{l,k} = E\left(\sum_{t=1}^{m} (dw_{i,j})_{l,t}\,(dw_{i,j})_{k,t}\right)$$

$$= \begin{cases} \sum_{t=1}^{m} \sigma_x^2 \sigma_y^2 & (l=k) \\ 0 & (l\neq k) \end{cases}$$

Therefore $Var[dw_{i,j}] = m\sigma_x^2\sigma_y^2 I = (n-k+1)\,\sigma_x^2\sigma_y^2 I$.

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

$$y_{1,1} = x_{1,1} = \max(x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}) \tag{17}$$

Then generalize to higher dimension and arbitrary max positions.

For the simplest case mentioned above.

$$dx_{i,j} = \frac{\partial \mathcal{L}}{\partial Y} \cdot \frac{\partial Y}{\partial x_{i,j}} = dY \frac{\partial Y}{\partial x_{i,j}}.$$

$$dx_{i,j} = \begin{cases} dy_{1,1} & (i=1, j=1) \\ 0 & (\text{else}) \end{cases}$$

For a more general case.

$y_{i,j} = \max \begin{pmatrix} x_{2i,2j} & x_{2i,2j+1} \\ x_{2i+1,2j} & x_{2i+1,2j+1} \end{pmatrix}$. Denote $\begin{pmatrix} x_{2i,2j} & x_{2i,2j+1} \\ x_{2i+1,2j} & x_{2i+1,2j+1} \end{pmatrix}$ as $\xi_{i,j}$.

$x_{i,j}$ will decide the value of $y_{l,k}$. ($l \in \{i/2, i/2+1\}$, $k \in \{j/2, j/2+1\}$)

$$\frac{\partial y_{l,k}}{\partial x_{i,j}} = \begin{cases} 1 & (l \in \{i/2, i/2+1\},\ k \in \{j/2, j/2+1\},\ x_{i,j} = \max \xi_{l,k}. \\ 0 & (\text{else}) \end{cases}$$

Hence $[d\, x_{i,j}]_{l,k} = \begin{cases} dy_{l,k} & (l \in \{i/2, i/2+1\},\ k \in \{j/2, j/2+1\},\ x_{i,j} = \max \xi_{l,k}. \\ 0 & (\text{else}) \end{cases}$

(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

Max pooling lays more stress on the distinct local maxima, which means can better extract local features.

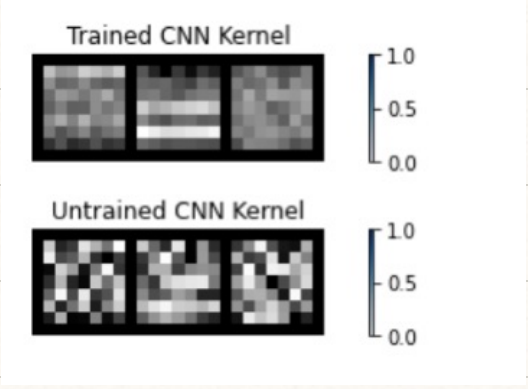
Average pooling considers all the pixels in a small region, and it is more resistent to noise.

4. Inductive Bias of CNNs (Coding Question)

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

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

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



Some weights are horizontally coherent.

(b) 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?

(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?

i) The untrained kernels have relatively random weights. The learned ones are more similar to edge-detector kernels, which usually contain horizontal or vertical lines.

ii). In this case, data is sufficient to fine-tune the weights, but when initialized with edge detectors, it could turn out they do not extract the ideal edge, causing misclassification.

(c) 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 20 images per class.

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

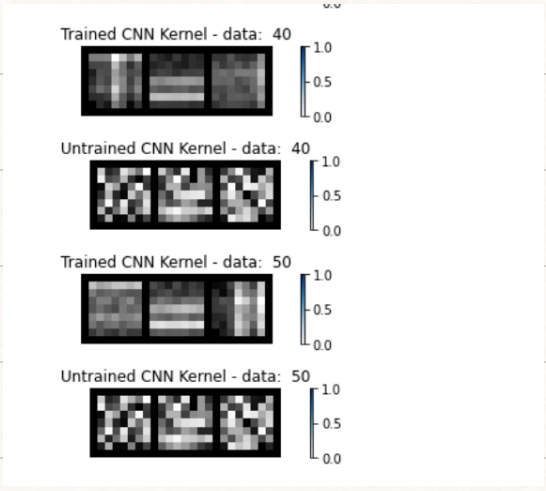
(iii) (optional) You might find that with the high number of epochs, validation loss of MLP is increasing whild validation accuracy increasing. How can we interpret this?

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

(v) How much more data is needed for MLP to get a competitive performance with CNN? Does MLP really generalize or memorize?

i) 100, 200, 300, 400. when training for 400 epochs we reached the expected accuracy.

ii)



The horizontal and vertical patterns are more clear.

iii). In the late stage of training process, the predicted probabilities are all driven to extreme (close to 0 or 1), and due to cross entropy is of the form $-\sum_i P(i) \log Q(i)$, where $P(i)$ is the true distribution and $Q(i)$ is the estimation, if one sample point is misclassified, which means $P(i)_j = 1$ and $Q(i)_j \to 0$, the loss will turn out extremely large. In this case, our loss is largely influenced by astypical sample pts.

iv).

v) In my test, when trained with 40 images, MLP reaches an acc of 88.0%, while CNN has 92% acc even when 10 images are given.

MLP does not generalize.

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

(i) Why do you think the confusion matrix looks like this? Why CNN misclassifies the images with edge to the images with no edge? Why MLP misclassifies the images with vertical edge to the images with horizontal edge and vice versa? (Hint: Visualize some of the images in the training and validation set.)

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

i) CNN: The kernel size is too large, and it may cause the model unable to capture smaller features.

MLP: The function for every layer is of the form $y = W \cdot x + b$, so when the domain of training set & validation set are different, the weights & input will be mismatched, failing to generalize.

ii) MLP does not consider spatial information, and it cannot tell apart whether the edge is horizontal or vertical.

(e) 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.)

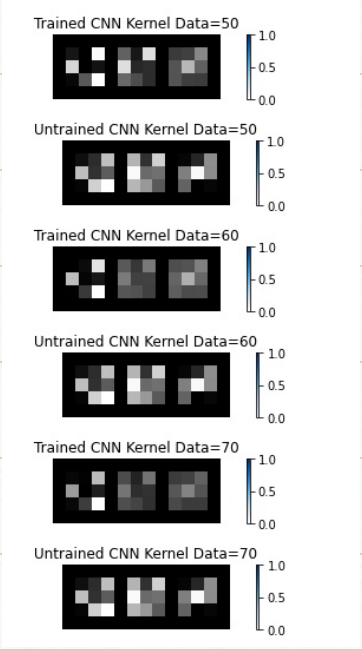
(ii) Assuming we are increasing kernel size of CNN. Does the validation accuracy increase or decrease? Why?

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

i) There is no strong correlation between adjacent pixels, which CNN pays attention to, but MLP does not take spatial info into consideration.

ii). The validation acc may not change since large perceptive field won't help when the correlations are rather weak.

iii)



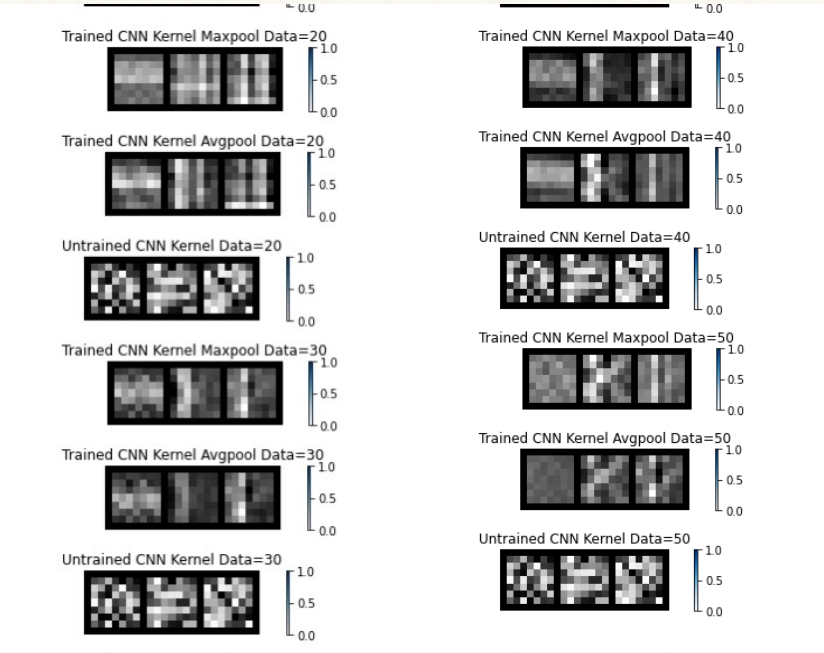
The trained kernel data is similar to that of untrained. cause there is no distinct correlation between adjacent pixels.

(f) 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?

i) ① Max pooling outperforms average pooling when trained with 10/20/30/40 images, but underperforms it when trained with 50 images.

② Max pooling is good at detecting edges, while average pooling can better keep the features from the background.


Trained CNN Kernel Maxpool Data=20
Trained CNN Kernel Avgpool Data=20
Untrained CNN Kernel Data=20
Trained CNN Kernel Maxpool Data=30
Trained CNN Kernel Avgpool Data=30
Untrained CNN Kernel Data=30
Trained CNN Kernel Maxpool Data=40
Trained CNN Kernel Avgpool Data=40
Untrained CNN Kernel Data=40
Trained CNN Kernel Maxpool Data=50
Trained CNN Kernel Avgpool Data=50
Untrained CNN Kernel Data=50
1.0
0.5
0.0

5. 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**?

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

(iii) After you load the model into memory, **what is the memory overhead (MB) of the CUDA context loaded with the model**?

i) 58164298 . 232.65192MB

ii). Persistence-M. 15360 MiB.

iii) 814 MB.

```
Wed Feb 22 23:58:11 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   66C    P0    29W /  70W |    814MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1834      C                                    811MiB |
+-----------------------------------------------------------------------------+
```

(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.

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

i)

```
========= Memory Profiling Results =========
SGD: 3192.0 MB
SGD_WITH_MOMENTUM: 3274.0 MB
ADAM: 3358.0 MB
```

ii) ADAM.

ADAM needs to store more intermediate variables when doing gradient decent.

$$L_{l_2}(\theta) = L(\theta) + \gamma||\theta||^2$$

SGD

$$\theta_t = \theta_{t-1} - \nabla L_{l_2}(\theta_{t-1}) = \theta_{t-1} - \nabla L(\theta_{t-1}) - \gamma\theta_{t-1}$$

SGDM

$$\theta_t = \theta_{t-1} - \lambda m_{t-1} - \eta(\nabla L(\theta_{t-1}) + \gamma\theta_{t-1})$$
$$m_t = \lambda m_{t-1} + \eta(\nabla L(\theta_{t-1}) + \gamma\theta_{t-1})\ ?$$
$$m_t = \lambda m_{t-1} + \eta(\nabla L(\theta_{t-1}))\ ?$$

Adam

$$m_t = \lambda m_{t-1} + \eta(\nabla L(\theta_{t-1}) + \gamma\theta_{t-1})\ ?$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(\theta_{t-1}) + \gamma\theta_{t-1})^2\ ?$$

(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?**

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

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

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

i) largest : 256

ii) batch_size = 16.

iii) batch_size = 16. when batch_size is too small, we cannot fully take advantage of the parallel processing ability of GPU. when BS is too large, it may reduce cache hit rate.

418-032 = 386

resnet18_4.csv

imestamp	epoch	memUsage	loss	accuracy
1677113032	1	1624	1.549990892	25.9765625
1677113075	2	1624	1.82569623	28.19824219
1677113118	3	1624	1.670931339	33.7890625
1677113160	4	1624	2.304392338	37.6953125
1677113203	5	1624	1.918154478	41.33300781
1677113246	6	1624	1.672734976	44.04296875
1677113289	7	1624	1.064380288	43.89648438
1677113332	8	1624	1.79573226	49.24316406
1677113375	9	1624	1.699944019	49.51171875
1677113418	10	1624	1.253587008	51.97753906

875-655 = 220.

resnet18_16.csv

timestamp	epoch	memUsage	loss	accuracy
1677113625	1	1706	2.048532724	28.90625
1677113654	2	1706	2.109718561	29.8828125
1677113682	3	1706	1.806398153	36.1328125
1677113709	4	1706	1.438007712	40.11230469
1677113737	5	1706	1.632109284	47.75390625
1677113765	6	1706	1.991657853	40.8203125
1677113792	7	1706	1.139603138	47.0703125
1677113820	8	1706	1.324549198	53.54003906
1677113847	9	1706	1.308345795	59.44824219
1677113875	10	1706	1.726740718	57.17773438

937-693 = 244

resnet18_64.csv

timestamp	epoch	memUsage	loss	accuracy
1677117693	1	2738	1.918797612	25.26855469
1677117719	2	2738	1.844422936	31.640625
1677117746	3	2738	1.589219809	38.01269531
1677117773	4	2738	1.606027246	31.22558594
1677117800	5	2738	1.398663521	38.25683594
1677117827	6	2738	1.520402551	38.69628906
1677117853	7	2738	1.310506105	53.58886719
1677117884	8	2738	1.274451971	55.24902344
1677117910	9	2738	1.314524293	55.90820313
1677117937	10	2738	0.9967767	56.25

588-353 = 235

resnet18_256.csv

timestamp	epoch	memUsage	loss	accuracy
1677118353	1	8744	2.135347366	10.69335938
1677118379	2	8744	1.925767541	12.52441406
1677118405	3	8744	1.669530034	31.59179688
1677118430	4	8744	1.642407179	35.30273438
1677118456	5	8744	1.551792741	26.22070313
1677118482	6	8744	1.465792775	42.60253906
1677118509	7	8744	1.348774314	34.61914063
1677118535	8	8744	1.274964452	43.01757813
1677118562	9	8744	1.133284092	46.36230469
1677118588	10	8744	1.052910924	51.02539063

iv).


```
====== Memory Usage for different batch sizes =======
4       : 1624.0 MB
16      : 1706.0 MB
64      : 2738.0 MB
256     : 8744.0 MB
```