DEPARTMENT OF ECE, UCLA
ECE 188: THE FOUNDATIONS OF COMPUTER VISION

INSTRUCTOR: Prof. Achuta Kadambi
TA: Howard Zhang, Shijie Zhou

NAME: Yaman Yucel
UID: 605704529

HOMEWORK 4

PROBLEM	TYPE	TOPIC	MAX. POINTS
1	Analytical	Machine Learning Basics	10
2	Coding	Training a Classifier	15
3	Interview Questions (Bonus)	Miscellaneous	15
4	Analytical	Generative adversarial networks	5

Motivation

The problem set gives you a basic exposure to machine learning approaches and techniques used for computer vision tasks such as image classification. You will train a simple classifier network on CIFAR-10 dataset using google colab. We have provided pytorch code for the classification question, you are free to use any other framework if that's more comfortable.

The problem set consists of two types of problems:

- analytical questions to solidify the concepts covered in the class, and
- coding questions to provide a basic exposure to building a machine learning classifier using pytorch.

This problem set also exposes you to a variety of machine learning questions commonly asked in job/internship interviews

Homework Layout

The homework consists of 3 problems in total, with subparts for each problem. There are 2 types of problems in this homework - analytical and coding. All the problems need to be answered in the Overleaf document. Make a copy of the Overleaf project, and fill in your answers for the questions in the solution boxes provided.

For the analytical questions you will be directly writing their answers in the space provided below the questions. For the coding problems you need to use the Jupyter notebooks (see the Jupyter notebook for each sub-part which involves coding). You are provided with 1 jupyter notebook for Problem 2. After writing your code in the Jupyter notebook you need to copy paste the same code in the space provided below that question on Overleaf. In some questions you are also required to copy the saved images (from Jupyter) into the solution boxes in Overleaf. For the classification question, upload the provided notebook to google colab, and change the runtime type to GPU for training the classifier on GPU. Refer to question 2 for more instructions/details.

Submission

You will need to make two submissions: (1) Gradescope: You will submit the Overleaf PDF with all the answers on Gradescope. (2) BruinLearn: You will submit your Jupyter notebooks (.ipynb file) with all the cells executed on BruinLearn.

Software Installation

You will need Jupyter to solve the homework. You may find these links helpful:

- Jupyter (https://jupyter.org/install)
- Anaconda (https://docs.anaconda.com/anaconda/install/)

1 Machine Learning Basics (10 points)

1.1 Calculating gradients (2.0 points)

A major aspect of neural network training is identifying optimal values for all the network parameters (weights and biases). Computing gradients of the loss function w.r.t these parameters is an essential operation in this regard (gradient descent). For some parameter w (a scalar weight at some layer of the network), and for a loss function L, the weight update is given by $w := w - \alpha \frac{\partial L}{\partial w}$, where α is the learning rate/step size.

Consider (a) w, a scalar, (b) $\mathbf{x}$, a vector of size $(m \times 1)$, (c) $\mathbf{y}$, a vector of size $(n \times 1)$ and (d) $\mathbf{A}$, a matrix of size $(m \times n)$. Find the following gradients, and express them in the simplest possible form (boldface lowercase letters represent vectors, boldface uppercase letters represent matrices, plain lowercase letters represent scalars):

- $z = \mathbf{x}^T\mathbf{x}$, find $\frac{dz}{d\mathbf{x}}$
- $z = Trace(\mathbf{A}^T\mathbf{A})$, find $\frac{dz}{d\mathbf{A}}$
- $z = \mathbf{x}^T\mathbf{A}\mathbf{y}$, find $\frac{\partial z}{\partial \mathbf{y}}$
- $\mathbf{z} = \mathbf{A}\mathbf{y}$, find $\frac{d\mathbf{z}}{d\mathbf{y}}$

You may use the following formulae for reference:

$$\frac{\partial z}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix}, \frac{\partial z}{\partial \mathbf{A}} = \begin{bmatrix} \frac{\partial z}{\partial A_{11}} & \frac{\partial z}{\partial A_{12}} & \cdots & \frac{\partial z}{\partial A_{1n}} \\ \frac{\partial z}{\partial A_{21}} & \frac{\partial z}{\partial A_{22}} & \cdots & \frac{\partial z}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z}{\partial A_{m1}} & \frac{\partial z}{\partial A_{m2}} & \cdots & \frac{\partial z}{\partial A_{mn}} \end{bmatrix}, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

1. The derivative of $x^T x = \sum_{i=1}^{m} (x_i)^2$ with respect to $\mathbf{x}$:

$$\frac{\partial \sum_{i=1}^{m} (x_i)^2}{\partial x_i} = 2x_i \Rightarrow \frac{\partial \mathbf{x}^T\mathbf{x}}{\partial \mathbf{x}} = \begin{bmatrix} 2x_1 \\ 2x_2 \\ \vdots \\ 2x_m \end{bmatrix} = 2\mathbf{x}$$

2. The derivative of $\text{tr}\left(\mathbf{A}^T\mathbf{A}\right) = \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2$ with respect to A:

$$\frac{\partial \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}{\partial a_{ij}} = 2a_{ij} \Rightarrow \frac{\partial \,\text{tr}\left(\mathbf{A}^T\mathbf{A}\right)}{\partial \mathbf{A}} = \begin{bmatrix} 2a_{11} & 2a_{12} & \ldots & 2a_{1n} \\ 2a_{21} & 2a_{22} & \ldots & 2a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 2a_{m1} & 2a_{m2} & \ldots & 2a_{mn} \end{bmatrix} = 2\mathbf{A}$$

3. The derivative of $\mathbf{x}^T\mathbf{A}\mathbf{y}$ with respect $\mathbf{y}$: Let $\mathbf{c} = \mathbf{A}^T\mathbf{x}$, then $\mathbf{x}^T\mathbf{A}\mathbf{y} = \mathbf{c}^T\mathbf{y}$

$$\mathbf{c}^T\mathbf{y} = \sum_{i=1}^{m}(c_i y_i) \Rightarrow \frac{\partial \sum_{i=1}^{m}(c_i y_i)}{\partial y_i} = c_i \Rightarrow \frac{\partial \mathbf{c}^T\mathbf{y}}{\partial \mathbf{y}} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \mathbf{c} = \mathbf{A}^T\mathbf{x}$$

4. The derivative of $\mathbf{A}\mathbf{y}$ with respect to $\mathbf{y}$

$$z_i = \sum_{k=1}^{n} a_{ik} y_k \Rightarrow \frac{\partial z_i}{\partial y_j} = a_{ij} \Rightarrow \frac{\partial \mathbf{z}}{\partial \mathbf{y}} = \mathbf{A}^{\mathbf{T}}$$

1.2 Deriving Cross entropy Loss (6.0 points)

In this problem, we derive the cross entropy loss for binary classification tasks. Let $\hat{y}$ be the output of a classifier for a given input x. y denotes the true label (0 or 1) for the input x. Since y has only 2 possible values, we can assume it follow a Bernoulli distribution w.r.t the input x. We hence wish to come up with a loss function $L(y, \hat{y})$, which we would like to minimize so that the difference between $\hat{y}$ and y reduces. A Bernoulli random variable (refresh your pre-test material) takes a value of 1 with a probability k, and 0 with a probability of $1-k$.

(i) Write an expression for $p(y|x)$, which is the probability that the classifier produces an observation $\hat{y}$ for a given input. Your answer would be in terms of $y, \hat{y}$. Justify your answer briefly.

For a single input(x): $p(y|x) = \hat{y}^y(1-\hat{y})^{1-y}$
For exactly m input(x): $p(y|x) = \prod_{i=1}^{m} \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$ since each input is i.i.d
Consider $y = 1$:
$p(y=1|x) = \hat{y}$ which is the probability of belonging to class 1 given the input x.
Consider $y = 0$:
$p(y=0|x) = 1-\hat{y}$ which is the probability of belonging to class 0 given the input x.
Therefore, the model follows Bernoulli distribution.

(ii) Using (i), write an expression for $\log p(y|x)$. $\log p(y|x)$ denotes the log-likelihood, which should be maximized.

For m input:
$log(p(y|x)) = log(\prod_{i=1}^{m} \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}) = \sum_{n=1}^{m} log(\hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i})$
$log(p(y|x)) = \sum_{i=1}^{m} y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)$
For single input:

$$log(p(y|x)) = ylog(\hat{y}) + (1-y)log(1-\hat{y})$$

(iii) How do we obtain $L(y,\hat{y})$ from $\log p(y|x)$? Note that $L(y,\hat{y})$ is to be minimized

Since loss needs to be minimized, we need to multiply log-likelihood with -1 to make maximization minimization.
For m input:
$L(y,\hat{y}) = \sum_{i=1}^{m} -y_i log(\hat{y}_i) - (1-y_i)log(1-\hat{y}_i)$
$L(y,\hat{y}) = -ylog(\hat{y}) - (1-y)log(1-\hat{y})$

1.3 Perfect Classifier (?) (2.0 points)

You train a classifier on a training set, achieving an impressive accuracy of 100 %. However to your disappointment, you obtain a test set accuracy of 20 %. For each suggestion below, explain why (or why not) if these suggestions may help improve the testing accuracy.

1. Use more training data
2. Add L2 regularization to your model
3. Increase your model size, i.e. increase the number of parameters in your model
4. Create a validation set by partitioning your training data. Use the model with highest accuracy on the validation set, not the training set.

1. IMPROVE: Use more training data
By using more training data, the model can generalize better, will have reduced sensitivity to outliers and individual samples. Also, it will learn better feature representations. Also, since the data contains noise term, using more data leads to improved regularization. Hence, it is possible to get better test accuracy.
2. IMPROVE: add L2 regularization to your model.
Adding L2 regularization to the loss enforces original loss to not converge to 0. If loss converges to zero, overfitting occurs and we get 100% training accuracy. Other than that, adding a L2 regularization term shrinks model parameters and forces solution to be more balanced and generalized. The model is discouraged from relying too heavily on specific features.Hence, it is possible to get better test accuracy.
3. DOES NOT IMPROVE: Increase your model size:
Increasing model complexity does not solve the problem of overfitting, on contrary it has the potential to increase overfitting. Therefore, test accuracy might get lower. By increasing model size, model can fit to noise term in data, which yields to bad performance for unseen data(test data).
4.IMPROVE: Create validation set by partitioning your training data. Use the model with highest accuracy on the validation set, not the training set.

Since model is not trained with the validation dataset, the data at validation dataset behaves unseen data. Therefore, we can tune the hyperparameters of the model with validation dataset to get better accuracy at testing procedure.Hence, it is possible to get better test accuracy.

2 Implementing an image classifier using PyTorch (15.0 points)

In this problem you will implement a CNN based image classifier in pytorch. We will work with the CIFAR-10 dataset. Follow the instructions in jupyter-notebook to complete the missing parts. For this part, you will use the notebook named PSET4_Classification. For training the model on colab gpus, upload the notebook on google colab, and change the runtime type to GPU.

2.1 Loading Data (2.0 points)

(i) Explain the function of transforms.Normalize() function (See the Jupyter notebook Q1 cell). How will you modify the arguments of this function for gray scale images instead of RGB images.
(ii) Write the code snippets to print the number of training and test samples loaded.

Make sure that your answer is within the bounding box.

(i)

```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

Above code, normalizes data in each channel using the all samples and the given mean and std values. Since we have 3 channels we have 3 values for the first input and 3 values for second input. First 3 inputs are given to set mean for each channel. Second 3 inputs are given to set std for each channel.
If we want to use gray scale images, above code will change to 1 channel version which is as follows:

```
transforms.Normalize((0.5), (0.5))
```

(ii)

```
print(train_data)
print(test_data)
```

Number of training samples: 50000
Number of testing sample: 10000
OUTPUT:
Dataset CIFAR10
Number of datapoints: **50000**
Root location: ./data
Split: Train
StandardTransform
Transform: Compose(
ToTensor()
Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
) Dataset CIFAR10
Number of datapoints: **10000**

Root location: ./data
Split: Test
StandardTransform
Transform: Compose(
ToTensor()
Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
)

2.2 Classifier Architecture (6.0 points)

(See the Jupyter Notebook) Please go through the supplied code that defines the architecture (cell Q2 in the Jupyter Notebook), and answer the following questions.

1. Describe the entire architecture. The description for each layer should include details about kernel sizes, number of channels, activation functions and the type of the layer.
2. What does the padding parameter control?
3. Briefly explain the max pool layer.
4. What would happen if you change the kernel size to 3 for the CNN layers without changing anything else? Are you able to pass a test input through the network and get back an output of the same size? Why/why not? If not, what would you have to change to make it work?
5. While backpropagating through this network, for which layer you don't need to compute any additional gradients? Explain Briefly Why.

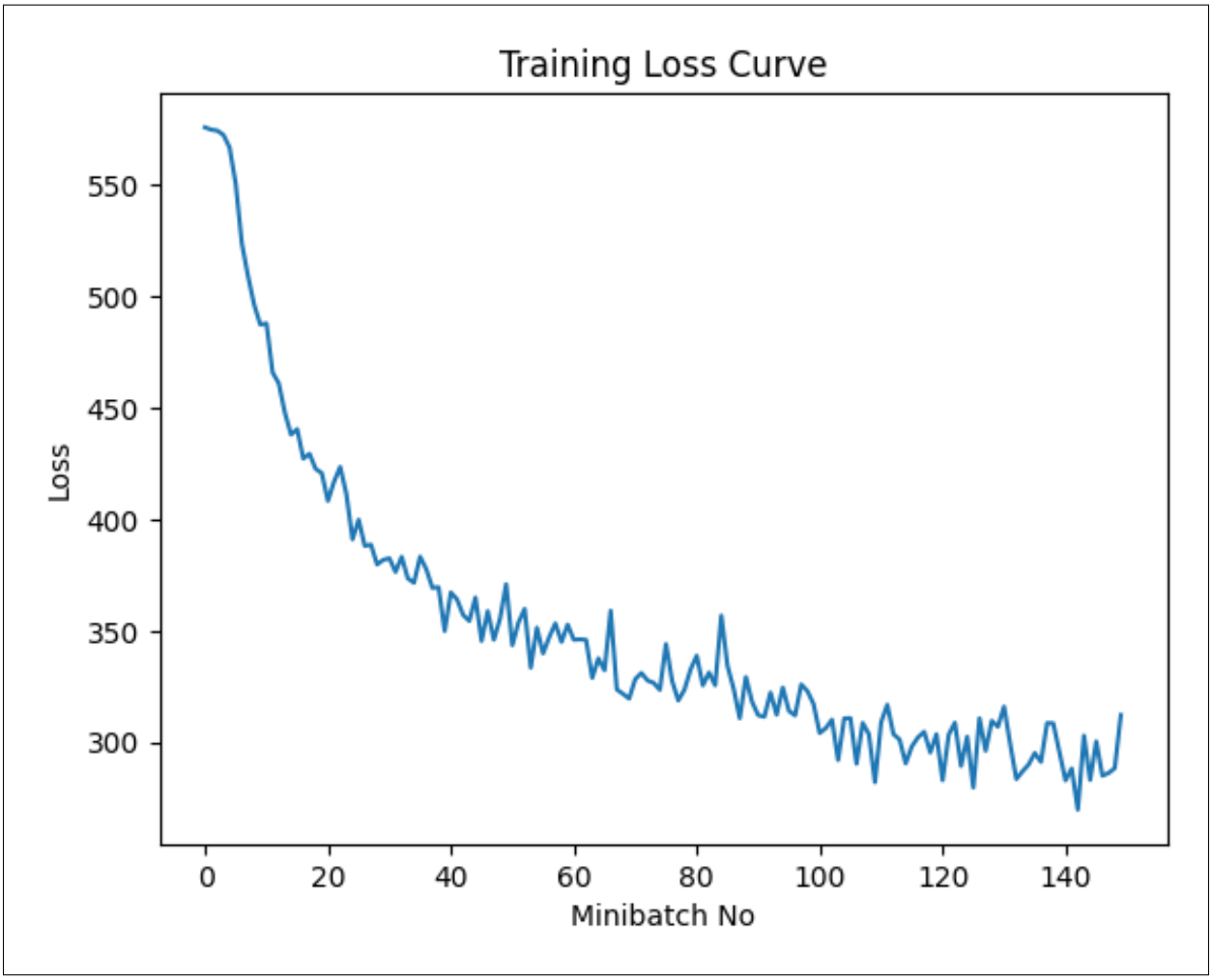
1. ANSWER: Architecture consists of a convolution layer, max pooling layer, convolution layer followed by 3 fully connected layers. Last layer is the output layer, where softmax is applied at the end. Using softmax we can minimize the cross entropy loss.
First convolution layer, $C_{in} = 3$, this makes sense since we have RGB channels for an image. $C_{out} = 6$, number of filters are 6, size of the filters(kernels) = 5, stride amount = 1, no padding, activation layer = ReLU, type convolution.
Max pooling layer, $C_{in} = 6$, $C_{out} = 6$, kernel size = 2, stride amount = 2, no padding, no activation layer, type max pooling.
Second convolution layer, $C_{in} = 6$, $C_{out} = 16$, kernel size= 5, stride amount = 1, no padding, ReLU activation, type convolution.
First fully connected layer, there are no channels, kernel size, stride and padding defined for fully connected layers. However, number of input neurons are 16*5*5 and number of output neurons are 120 for the first FCN. Activation function is ReLU. Type of the layer is fully connected layer.
Second fully connected layer, number of input neurons are 120 and number of output neurons are 84 for the second FCN. Activation function is ReLU. Type of the layer is fully connected layer.

Third and output fully connected layer, number of input neurons are 84 and number of output neurons are 10. Activation function is ReLU. Type of the layer is fully connected layer. Different from other networks output of this layer is connected to softmax function to find probabilities to belonging to classes.

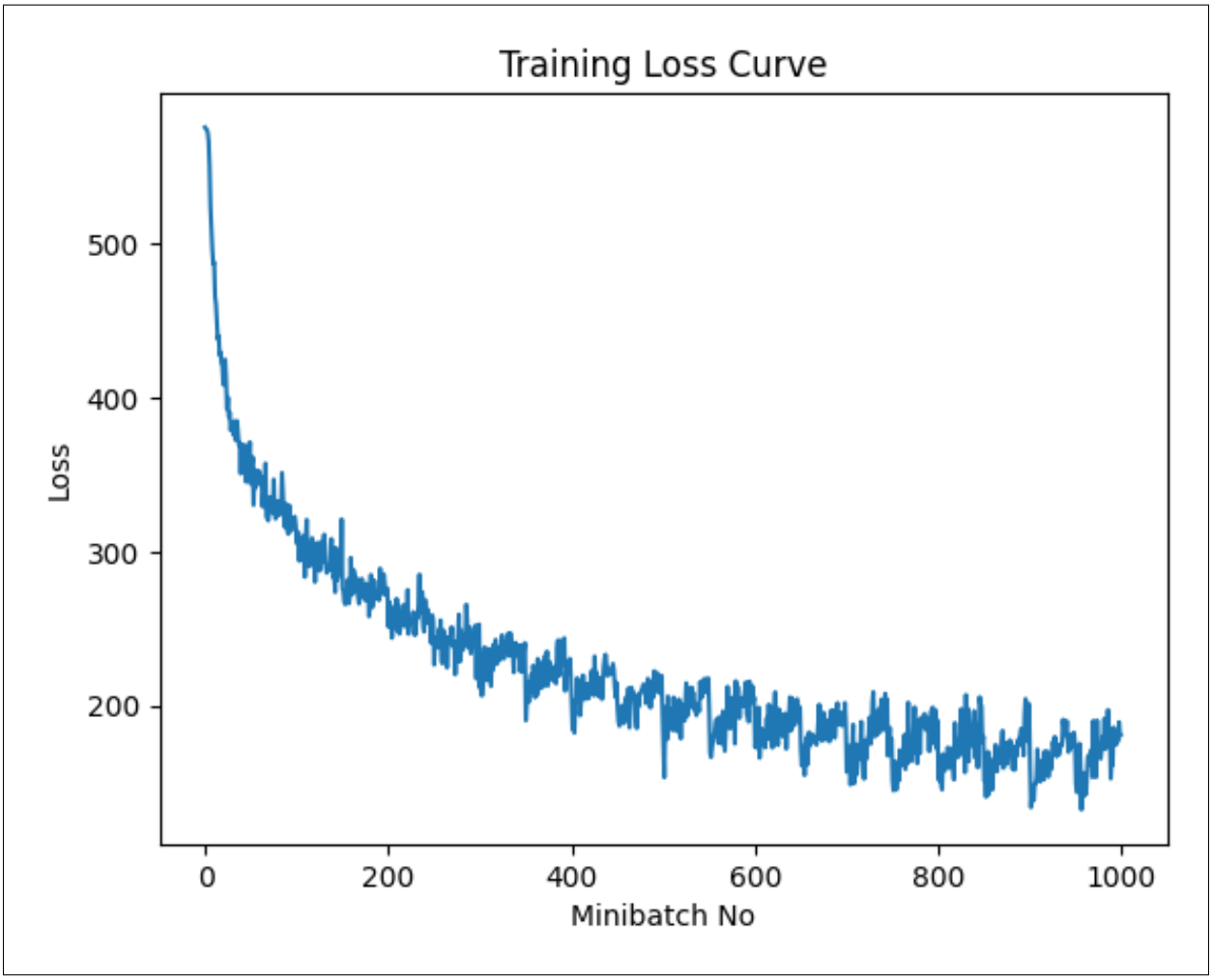
2. ANSWER: Padding controls the amount of padding applied to the input. Amount of "padding" is added to both 4 sides of the image. Therefore input size increases by 2*padding for each dimension.
3. ANSWER: Max pool layer is similar to applying convolution layer. In this layer, rather doing cross correlation with the kernel. Kernel is a nonlinear function and applies maximum. Therefore, rather than multiplying all numbers with the kernel, kernel takes maximum of the all numbers seen by the kernel. The dimension of the max pool kernel affects the number of fields received in the layer.
4. ANSWER: Receptive field of the convolutional layer increases, however this leads to increase in the size of the output of convolutional layer. If we pass a test input, the output will be different since output sizes are related to input sizes with the following formulation. $\frac{w-k+2pad}{s}+1$. This formula is same for width and length. k is the kernel size, w is the input size, pad is the padding amount and s is stride. If we choose k as 3 rather than 5, w - k will increase and output will be larger in size. However, it is possible to get same output size by adjusting padding and stride amount. $\frac{w-k_1+2pad_1}{s_1}+1=\frac{w-k_2+2pad_2}{s_2}+1$. We need to solve this equation to find integer values for pad_2 and s_2, note in our example $k_1=5, k_2=3, pad_1=0, s_1=1$.
5. ANSWER: We do not need to compute additional gradients for max pooling layer since it does not have any learnable parameters. If it does not have any learnable parameters, we do not need to update them. However, we need to compute gradients for all convolutional and fully connected layers.

2.3 Training the network (3.0 points)

(i) (See the Jupyter notebook.) Complete the code in the jupyter notebook for training the network on a CPU, and paste the code in the notebook. Train your network for 3 epochs. Plot the running loss (in the notebook) w.r.t epochs.



(ii) (See the Jupyter notebook.) Modify your training code, to train the network on the GPU. Paste here the lines that need to be modified to train the network on google colab GPUs. Train the network for 20 epochs



(iii) Explain why you need to reset the parameter gradients for each pass of the network

PyTorch accumulates the gradients on subsequent backward passes. This accumulating behavior is convenient while training RNNs or when we want to compute the gradient of the loss summed over multiple mini-batches. So, the default action has been set to accumulate the gradients on every loss.backward() call. However, since we are using CNN, we do not want accumulating gradients. Gradients are recalculated from scratch in each step. Because of this, when we start our training loop, ideally we should zero out the gradients so that you do the parameter update correctly. Otherwise, the gradient would be a combination of the old gradient, which you have already used to update your model parameters and the newly-computed gradient.

2.4 Testing the network (4.0 points)

(i) (See the jupyter-notebook) Complete the code in the jupyter-notebook to test the accuracy of the network on the entire test set.

```
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
acc  = correct * 100/total
print('Accuracy of the network on the 10000 test images: %d %%' % (acc))
```

OUTPUT: Accuracy of the network on the 10000 test images: 59%

(ii) Train the network on the GPU with the following configurations, and report the testing accuracies and running loss curves -

- Training Batch Size 4, 20 training epochs
- Training Batch Size 4, 5 epochs
- Training Batch Size 16, 5 epochs
- Training Batch Size 16, 20 epochs

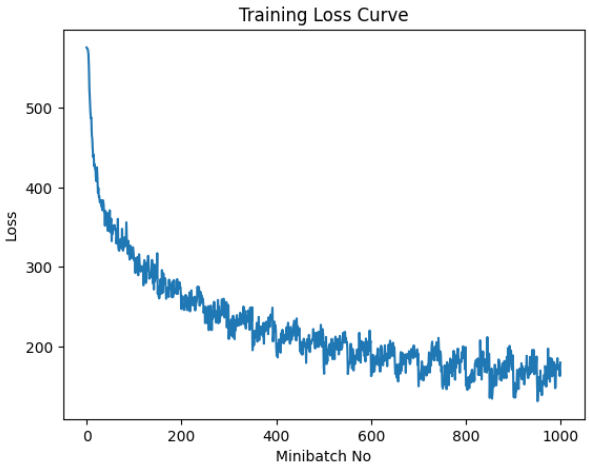


Figure 1: Training Batch Size 4, 20 training epochs, Test accuracy: **61%**

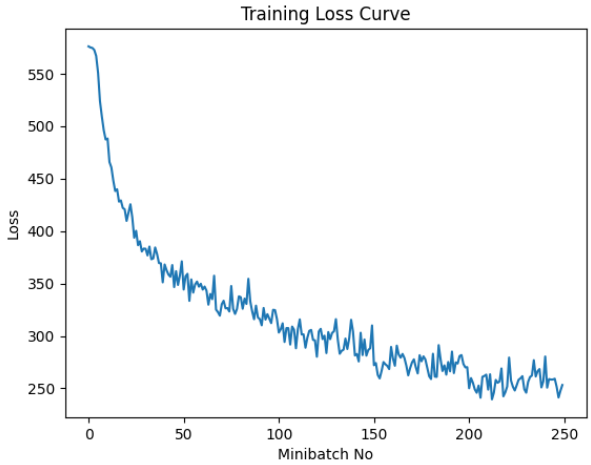


Figure 2: Training Batch Size 4, 5 epochs, Test accuracy:**60%**

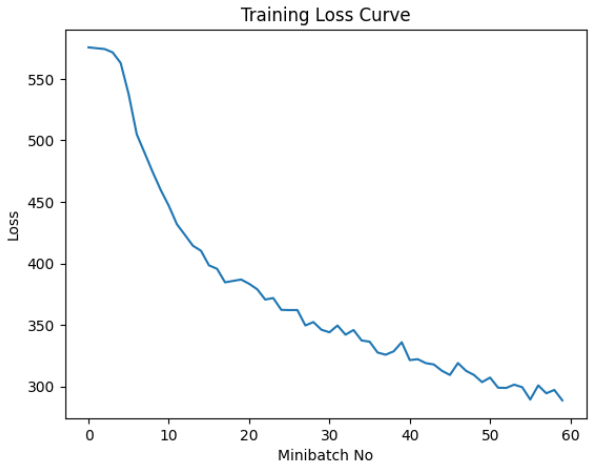


Figure 3: Training Batch Size 16, 5 epochs, Test accuracy: **58%**

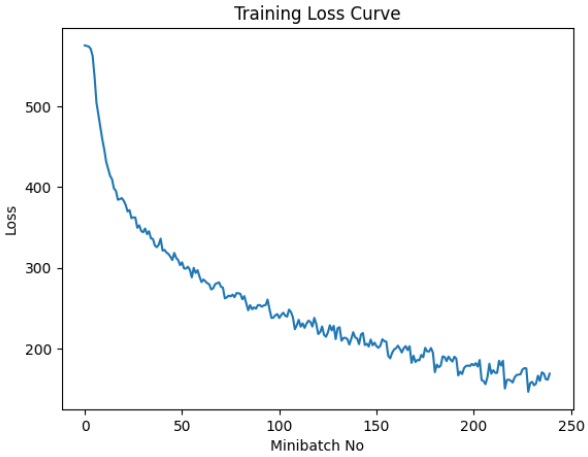


Figure 4: Training Batch Size 16, 20 epochs, Test accuracy: **64%**

(iii) Explain your observations in (ii)

1. Batch Size:
Batch size is related with the true estimation of the loss at that step. If the batch size is larger, the loss is estimated better but computation time increases. If the batch size is smaller, the loss is noisy due to estimation error. This estimation error can be easily from Figure 1 and 2, curve is oscillating while converging to a minimum loss. Curve in Figure 3 and 4 are smoother than Figure 1 and 2.
2. Number of Epochs:
Increasing number of epochs yields to increase in training time, but we can obtain a more correct converged loss value. Note that, number of epochs should be optimized such that model does not underfit nor overfit. Increasing too much number of epochs will make model to overfit to training dataset, since loss is recalculated multiple times with the same data. Using small number of epochs will lead to unconverged loss, therefore model does not train sufficiently.
Result: In our case, it seems that the best model is batch size 16, number of epochs 20 since it has the best test accuracy with **64%**. However, we should select model using this technique since this technique will fail for unseen data. Using this technique, we overfit to hyperparameters. Cross validation technique is commonly used to overcome this problem. The increase in accuracy is not sufficient, one can use dropout and batch normalization technique to increase generalization to dataset. Data augmentation is an another technique that is simple and efficient to obtain generalization.

3 Interview Questions (15 points)

3.1 Batch Normalization (4 points)

Explain
(i) Why batch normalization acts as a regularizer.
(ii) Difference in using batch normalization at training vs inference (testing) time.

(i) (i.1)Internal Covariate Shift:
Internal Covariate Shift refers to the phenomenon where the distribution of the intermediate activations of the network's layers changes during training as the parameters of the preceding layers are updated. This can make training more difficult and lead to slower convergence or even unstable training. Batch normalization addresses this issue by normalizing the activations within each mini-batch during training. By subtracting the batch mean and dividing by the batch standard deviation, batch normalization helps to stabilize and normalize the distributions of the intermediate activations. This normalization reduces the covariate shift, making the network more robust to changes in input distributions and improving training efficiency. Also, similar to weight decay, prevents exploding gradients which harm the training process.

(i.2)Regularization through Noise Injection:
Batch normalization introduces noise into the network by estimating the batch mean and variance. This noise injection acts as a form of regularization during training. The introduced noise can prevent the network from fitting the training data too closely and can reduce overfitting.

(ii) During training, batch normalization technique is used to find the mean and standard deviation of the activations that are computed at that step. Then these mean and standard deviation are scaled and shifted by learnable parameters. Therefore, the model also learns the distribution of the activations at the training step.
During testing, gamma and beta parameters that are previously learned in the training are directly used to normalize data points, therefore using the stored statistics, model knows the distribution of testing beforehand. This leads to increase in performance and stability.
In brief, we learn distribution at the training to normalize. We apply normalization at the testing, there is no adjustment to parameters in testing.

3.2 CNN filter sizes (4 points)

Assume a convolution layer in a CNN with parameters $C_{in} = 32$, $C_{out} = 64, k = 3$. If the input to this layer has the parameters $C = 32, H = 64, W = 64$.

(i) What will be the size of the output of this layer, if there is no padding, and stride = 1
(ii) What should be the padding and stride for the output size to be $C = 64, H = 32, W = 32$

(i)Size of the output layer: 64-3+1 x 64-3+1 x 64 = 62x62x64, H,W,C
(ii) $\frac{w-k+2pad}{s} + 1 = 32 = \frac{64-3+2pad}{2} + 1 \Rightarrow p = 16, s = 3$

3.3 L2 regularization and Weight Decay (4 points)

Assume a loss function of the form $L(y,\hat{y})$ where y is the ground truth and $\hat{y} = f(x,w)$. x denotes the input to a neural network (or any differentiable function) $f()$ with paramters/weights denoted by w. Adding $L2$ regularization to $L(y,\hat{y})$ we get a new loss function $L'(y,\hat{y}) = L(y,\hat{y}) + \lambda w^T w$, where λ is a hyperparameter. Briefly explain why $L2$ regularization causes weight decay. Hint: Compare the gradient descent updates to w for $L(y,\hat{y})$ and $L'(y,\hat{y})$. Your answer should fit in the given solution box.

$w := w - \alpha \frac{\partial L(y,\hat{y})}{\partial w}$, where α is the learning rate/step size.
Suppose we use the new loss function.
$w := w - \alpha \frac{\partial L'(y,\hat{y})}{\partial w}$
$w := w - \alpha \frac{\partial L(y,\hat{y})}{\partial w} - \alpha \frac{\partial \lambda w^T w}{\partial w}$
$w := w - \alpha \frac{\partial L(y,\hat{y})}{\partial w} - 2\alpha\lambda w$
$w := (1 - 2\alpha\lambda)w - \alpha \frac{\partial L(y,\hat{y})}{\partial w}$
Therefore, the term $(1 - 2\alpha\lambda w)$ causes weight decay and the decay is controlled by the λ and α. The better explanation will be given when the gradient computed is zero. Although the gradient is zero, the new update will shrink the weight by $(1 - 2\alpha\lambda w)$.

3.4 Why CNNs? (3 points)

Give 2 reasons why using CNNs is better than using fully connected networks for image data.

1) Local and Spatial Connectivity
CNNs are specifically designed to exploit the spatial structure and local correlations present in images. Unlike FCNs, which treat input data as a flattened vector, CNNs preserve the spatial relationship between pixels. By using convolutional layers, CNNs focus on local receptive fields, capturing local patterns and features which allows CNNs to work better for image data.

2)Parameter Sharing/ Sparsity:
Parameter sharing means that the same set of weights is applied across different regions of the input, reducing the number of parameters to be learned. This makes CNNs more efficient and effective in modeling image data, as they can learn translation-invariant features and capture spatial hierarchies. Parameter sharing is important since computation power can be easily spent when we are dealing with image data.

4 GAN (Bonus) (5.0 points)

4.1 Understanding GANs- Loss function (2.0 points)

Mathematically express the overall GAN loss function being used. For a (theoretically) optimally trained GAN: (a) what is the ideal behavior of the discriminator, and (b) what is the value of the overall loss function?

$L(G,D) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_{\text{noise}}}[\log(1-D(G(z)))]$
G is the generator network
D is the discriminator network
x is drawn from the real data samples
z is drawn from a normal distribution
D(x) is the output probability of the discriminator network given that the sample is real
D(G(x)) is the output probability of the discriminator network given that the sample is generated by generator network
Generator tries to maximize the function, whereas discriminator tries to minimize the loss function, therefore optimization of the loss function is different from regular neural networks.
a) The converged and ideal behavior of the discriminator will be halfly identifying the generated samples as fake. Therefore, output probability of discriminator will be 0.5 regardless of the input.
b) If the GAN has converged and generator started to produce realistic examples, then loss function should be near to -log(4) = -1.386, this loss is for one sample from data and one sample from noisy distribution. This shows that the generator is able to fool the discriminator and discriminator can not perform well. log(0.5) + log(1-0.5) = 2log(0.5) = -log(4)

4.2 Understanding GANs- Gradients (2.0 points)

Assume that you are working with a GAN having the following architecture:

Generator: Input: $\mathbf{x}$ shape $(2,1)$ $\rightarrow$ Layer: $\mathbf{W_g}$ shape $(5,2)$ $\rightarrow$ ReLU $\rightarrow$ Output: $\mathbf{y}$ shape $(5,1)$
Discriminator: Input: $\mathbf{z}$ shape $(5,1)$ $\rightarrow$ Layer: $\mathbf{W_d}$ shape $(1,5)$ $\rightarrow$ Sigmoid $\rightarrow$ Output: b shape $(1,1)$

Therefore, the generator output is given by, $\mathbf{y} = ReLU(\mathbf{W_g x})$, and the discriminator output is given by $b = Sigmoid(\mathbf{W_d z})$. Express the gradient of the GAN loss function, with respect to the weight matrices for the generator and discriminator.

You may use the following information:

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

Hint: Remember that the input here is a vector, not an image.

Since we have two different networks, backpropagation should be done two times, one for generator network and one for discriminator network.
However, there will be no contribution from the first part of the loss to the generator network, therefore one can write the following.
$\frac{\partial L(G,D)}{\partial W_g} = \frac{\partial log(1-D(G(z)))}{\partial W_g} = \frac{\partial L_g}{\partial W_g}$
Also for discriminator network, note we do not discard any parts.
$\frac{\partial L(G,D)}{\partial W_d} = \frac{\partial log(D(x))+log(1-D(G(z)))}{\partial W_d}$

One can rewrite the loss of generative network as computation graph:
$L_g = log(1-b_g)$
$b_g = \sigma(a_1)$, $sigmoid(x) = \sigma(x)$
$a_1 = W_d z$
$z = ReLU(a_2)$
$a_2 = W_g x$
Then, backpropagation can be easily applied:
$\frac{\partial L_g}{\partial b_g} = -\frac{1}{1-b_g}$
$\frac{\partial L_g}{\partial a_1} = \frac{\partial b_g}{\partial a_1}\frac{\partial L_g}{\partial b_g} = \sigma(a_1)(1-\sigma(a_1))(-\frac{1}{1-b_g})$
$\frac{\partial L_g}{\partial z} = \frac{\partial a_1}{\partial z}\frac{\partial L_g}{\partial a_1} = (W_d)^T\sigma(a_1)(1-\sigma(a_1))(-\frac{1}{1-b_g})$
$\frac{\partial L_g}{\partial a_2} = \frac{\partial z}{\partial a_2}\frac{\partial L_g}{\partial z} = \mathbb{1}(z>0)\odot((W_d)^T\sigma(a_1)(1-\sigma(a_1))(-\frac{1}{(}1-b_g)))$ or
$diag(\mathbb{1}(z>0))((W_d)^T\sigma(a_1)(1-\sigma(a_1))(-\frac{1}{1-b_g}))$
We can apply tensor trick to find derivative of a vector respect to a matrix.
$\frac{\partial L_g}{\partial W_g} = \frac{\partial a_2}{\partial W_g}\frac{\partial L_g}{\partial a_2} = \frac{\partial L_g}{\partial a_2}x^T = diag(\mathbb{1}(z>0))((W_d)^T\sigma(a_1)(1-\sigma(a_1))(-\frac{1}{1-b_g}))x^T$
$\frac{\partial L}{\partial W_g} = \frac{\partial L_g}{\partial W_g} = diag(\mathbb{1}(z>0))((W_d)^T\sigma(a_1)(1-\sigma(a_1))(-\frac{1}{1-b_g}))x^T$
Similar procedure can be applied to find derivative of loss with respect to W_d:

$L = L_d + L_g$
$L_d = log(b_d)$
$b_d = \sigma(c_1)$
$c_1 = W_d z$

$L_g = log(1-b_g)$
$b_g = \sigma(c_2)$
$c_2 = W_d y$
$\frac{\partial L_d}{\partial b_d} = \frac{1}{b_d}$
$\frac{\partial L_d}{\partial c_1} = \frac{\partial b_d}{\partial c_1}\frac{\partial L_d}{\partial b_d} = \sigma(c_1)(1-\sigma(c_1))\frac{1}{b_d}$
$\frac{\partial L_d}{\partial W_d} = \frac{\partial c_1}{\partial W_d}\frac{\partial L_d}{\partial c_1} = z^T\sigma(c_1)(1-\sigma(c_1))\frac{1}{b_d}$

$\frac{\partial L_g}{\partial b_g} = -\frac{1}{1-b_g}$
$\frac{\partial L_g}{\partial c_2} = \frac{\partial b_g}{\partial c_2}\frac{\partial L_g}{\partial b_g} = \sigma(c_2)(1-\sigma(c_2))(-\frac{1}{1-b_g})$
$\frac{\partial L_g}{\partial W_d} = \frac{\partial c_2}{\partial W_d}\frac{\partial L_g}{\partial c_2} = y^T\sigma(c_2)(1-\sigma(c_2))(-\frac{1}{1-b_g})$

$\frac{\partial L}{\partial W_d} = \frac{\partial L_d}{\partial W_d} + \frac{\partial L_g}{\partial W_d} = y^T\sigma(c_2)(1-\sigma(c_2))(-\frac{1}{1-b_g}) + z^T\sigma(c_1)(1-\sigma(c_1))\frac{1}{b_d}$
$\frac{\partial L}{\partial W_d} = ReLU(W_g x)^T\sigma(c_2)(1-\sigma(c_2))(-\frac{1}{1-b_g}) + z^T\sigma(c_1)(1-\sigma(c_1))\frac{1}{b_d}$
where x is drawn randomly from normal distribution and z is the real data.

4.3 Understanding GANs- Input distributions (1.0 points)

While training the GAN, the input is drawn from a normal distribution. Suppose in a hypothetical setting, each time the input is chosen from a different, randomly chosen probability distribution. How would this affect the training of the GAN? Justify your answer mathematically.

Training of GAN will significantly change if the input is not drawn from a normal distribution. Firstly, by using the normal distribution, generator can learn to generate output using the consistent pattern of input. It would be hard for generator to learn the distribution of the input to generate realistic samples. Secondly, GANs are difficult to train due to instability of the loss function. The instability will be introduced to training if the data is not drawn from a normal distribution, hence generator can only produce limited range of samples or vanishing gradients which hinder the training process. Lastly, GAN needs to converge to Nash Equilibrium to accurately generate realistic examples, the input randomly chosen will disrupt the convergence to equilibrium. However, consistent and easily learnable distribution can be used as input, we are not restricted to use normal distribution.