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

**INSTRUCTOR:** Prof. Achuta Kadambi
**TA:** Rishi Upadhyay

**NAME:** Your Name
**UID:** Your UID

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 from here `https://www.overleaf.com/read/pqksjrzjrckj#e991de`, 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 from here: https://colab.research.google.com/drive/1ciRDVvVRyIx5c48yF21BWf-ttnOtx83k?usp=sharing(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

Submission will be done via Gradescope. You will need to submit three things: (1) this PDF with answers filled out (from Overleaf), (2) Your .ipynb file, (3) a PDF printout of your .ipynb file with all cells executed. You do not need to create a folder, you will be able to upload all three files directly to gradescope.

## 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 $\alpha$ 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:

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

We use the hints to derive the solutions:

1. $z = \mathbf{x}^T\mathbf{x}$ is a scalar, and we want $\frac{dz}{d\mathbf{x}}$, which is a column vector. Using the chain rule and the fact that $\frac{d\mathbf{x}^T}{d\mathbf{x}} = \mathbf{I}$ (identity matrix), we get:

$$\frac{dz}{d\mathbf{x}} = \frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{x}) = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \\ \vdots \\ 2x_m \end{bmatrix} = 2\mathbf{x}$$

2. $z = \text{Trace}(\mathbf{A}^T\mathbf{A})$ is a scalar, and we want $\frac{dz}{d\mathbf{A}}$, which is a matrix of the same size as $\mathbf{A}$. Using

the chain rule and the fact that $\frac{d\text{Trace}(\mathbf{B})}{d\mathbf{B}} = \mathbf{I}$ (identity matrix of the same size as $\mathbf{B}$), we get:

$$\frac{dz}{d\mathbf{A}} = \frac{d}{d\mathbf{A}}(\text{Trace}(\mathbf{A}^T\mathbf{A})) = \begin{bmatrix} 2A_{11} & 2A_{12} & \dots & 2A_{1n} \\ 2A_{21} & 2A_{22} & \dots & 2A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 2A_{m1} & 2A_{m2} & \dots & 2A_{mn} \end{bmatrix} = 2\mathbf{A}$$

3. $z = \mathbf{x}^T\mathbf{A}\mathbf{y}$ is a scalar, and we want $\frac{\partial z}{\partial \mathbf{y}}$, which is a column vector of the same size as $\mathbf{y}$. Using the chain rule and the fact that $\frac{\partial \mathbf{x}^T\mathbf{A}}{\partial \mathbf{y}} = \mathbf{0}$, we get:

$$\frac{\partial z}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}}(\mathbf{x}^T\mathbf{A}\mathbf{y}) = \mathbf{A}^T\mathbf{x}$$

4. $\mathbf{z} = \mathbf{A}\mathbf{y}$ is a column vector, and we want $\frac{d\mathbf{z}}{d\mathbf{y}}$, which is a matrix of size $(m \times n)$. Using the chain rule and the fact that $\frac{\partial \mathbf{A}}{\partial \mathbf{y}} = \mathbf{0}$, we get:

$$\frac{d\mathbf{z}}{d\mathbf{y}} = \frac{d}{d\mathbf{y}}(\mathbf{A}\mathbf{y}) = \mathbf{A}$$

### 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.

We recall from prob theory that a random variable $Y$ that follows a Bernoulli distribution, the probability mass function (PMF) is:

$$P(Y = y) = p^y(1-p)^{1-y}, \quad y \in \{0, 1\}$$

Applying this to binary classification, where $\hat{y}$ is the predicted probability that $Y = 1$ given input $x$, and $y$ is the actual outcome (true label), the PMF for observing $y$ can be expressed as:

$$p(y|x) = \hat{y}^y(1-\hat{y})^{1-y}$$

Based on the derivation of the PMF from prob tehory, we believe this answer is correct

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

Given the probability $p(y|x) = \hat{y}^y(1-\hat{y})^{1-y}$ from part (i), we can take the logarithm of both sides to obtain the log-likelihood, $\log p(y|x)$. Applying logarithmic properties, we get:

$$\log p(y|x) = \log[\hat{y}^y(1-\hat{y})^{1-y}]$$

Using the property that $\log(ab) = \log a + \log b$, we can separate the terms:

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

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

I base my asnwer on my previous knowledge of optimization and machine learning, primarily on binary classifcaiton.

We cuurently have a liklihood function but traditional optimization frameworks minimize, so to obtain the loss function $L(y,\hat{y})$, we simply take the negative of the log-likelihood which gets us a loss function called the cross-entropy loss. We showcase below.

Given the log-likelihood expression:

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

The cross-entropy loss $L(y,\hat{y})$ is defined as the negative of this log-likelihood:

$$L(y,\hat{y}) = -\log p(y|x) = -[y\log(\hat{y}) + (1-y)\log(1-\hat{y})]$$

### 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. This may help.** Adding more training data can help improve the model's generalization ability. I think I once heard you can improve ML models easily if you add more samples to the dataset.

**2. This may also help.** We are taught in ML classes that L2 regularization aka ridge (regression? regularization?) is used to penalize model weights from every becoming too large. This helps models not overfit onto data and generalize.

**3. This will not help** Since our model is not generalizing well, adding more parameters will cause the model to memorize and generalize less. We are often taught that simple models are better.

**4. This will help.** A validation set generally helps to tune hyperparameters and prevent overfitting. It provides an unbiased estimate of the model's performance on unseen data, facilitating the selection of the best-performing model. By testing the method after each iteration of training, we can see the model improvement or its convergence on a local minima.

## 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)** The `transforms.Normalize()` function in PyTorch's torchvision module is utilized to normalize image pixel values so that they have a predefined mean and standard deviation for each channel. This normalization aids in model training efficiency and faster convergence. For RGB images, it requires two tuples representing the mean (`mean=[mean_R, mean_G, mean_B]`) and standard deviation (`std=[std_R, std_G, std_B]`) for each of the R, G, and B channels. For grayscale images, which have a single channel, the function's arguments should be modified to contain only one value for the mean and one for the standard deviation, i.e., `transforms.Normalize(mean=[mean_gray], std=[std_gray])`.

**(ii)** To print the number of training and test samples in the CIFAR-10 dataset, one can use the following code snippets:

```
print(f"Number of training samples: {len(train_loader)}")
print(f"Number of test samples: {len(test_loader)}")
```

### 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. Architecture Description:**

- `conv1`: A 2D convolutional layer with input channels=3 (RGB image), output channels=6, kernel size=$5 \times 5$, stride=1, and no padding.
- `pool`: A max pooling layer with kernel size=$2 \times 2$ and stride=2.
- `conv2`: A 2D convolutional layer with input channels=6, output channels=16, kernel size=$5 \times 5$, stride=1, and no padding.
- `pool`: Another max pooling layer with kernel size=$2 \times 2$ and stride=2.
- `fc1`: A fully connected linear layer with input size=$16 \times 5 \times 5$ (from the previous layer) and output size=120. No activation function is applied here.
- `fc2`: A fully connected linear layer with input size=120 and output size=84. No activation function is applied here.
- `fc3`: A fully connected linear layer with input size=84 and output size=10 (assuming 10 classes for classification). No activation function is applied here.
- The ReLU activation function is applied after `conv1`, `conv2`, `fc1`, and `fc2` layers.

**2. Padding Parameter:** The padding parameter in convolutional layers controls how the input is padded with zeros around the edges before applying the convolution operation. This is done to preserve the spatial dimensions (height and width) of the input after convolution. In the given code, the padding parameter is set to 0 for both `conv1` and `conv2` layers, which means no padding is applied.

**3. Max Pool Layer:** The max pooling layer is a downsampling operation that reduces the spatial dimensions of the input. It applies a sliding window (kernel) of a specified size and selects the maximum value within that window. In this code, the `pool` layer uses a kernel size of $2 \times 2$ and a stride of 2, which means it selects the maximum value from non-overlapping $2 \times 2$ regions of the input and downsamples it by a factor of 2 in both height and width dimensions. This operation helps to capture the most significant features while reducing the computational complexity.

**4. Changing Kernel Size and Input/Output Sizes:** If you change the kernel size to 3 for the CNN layers (`conv1` and `conv2`) without changing anything else, you will encounter an issue with the input/output sizes. The current input size for `fc1` is calculated based on the output size of `conv2`, which is $16 \times 5 \times 5$. However, with a kernel size of 3, the output size of `conv2` will be different, and the input size for `fc1` will no longer match.

To pass a test input through the network and get an output of the same size (10, assuming 10 classes), you would need to adjust the input size of `fc1` to match the output size of `conv2` with the new kernel size. You can do this by modifying the `view` operation in the `forward` method to match the output size of `conv2`.

**5. Layer Without Additional Gradients:** During backpropagation through this network, you don't need to compute additional gradients for the max pooling layer (`pool`). The reason is

that the max pooling operation is a non-linear operation that does not involve any trainable parameters (weights or biases). It is a fixed operation that selects the maximum value from a local region of the input.

During backpropagation, the gradients for the layers before the max pooling layer (`conv1` and `conv2`) can be computed using the gradients from the subsequent layers. However, the max pooling layer itself does not require any additional gradient computations since it does not have trainable parameters.

### 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.

```
### Complete the code in the training box

## for reproducibility
torch.manual_seed(7)
np.random.seed(7)

## Instantiating classifier
net = Net().cuda()

## Defining optimizer and loss function
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)


## Defining Training Parameters

num_epochs = 3 # 2 for CPU training, 10 for GPU training
running_loss_list = [] # list to store running loss in the code below
average_loss_per_epoch = []
for epoch in range(num_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    total_batches = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        #============================================#
        optimizer.zero_grad()
```

```
        # Forward pass
        outputs = net(inputs.cuda()) # Ensure inputs are transferred to
         ↪ GPU with .cuda()
        # Calculate loss
        loss = criterion(outputs, labels.cuda()) # Ensure labels are
         ↪ transferred to GPU with .cuda()
        # Backward pass
        loss.backward()
        # Optimize
        optimizer.step()
        #================================================#
        # print statistics
        running_loss += loss.cpu().item()
        total_batches += 1
        if i % 250 == 249:    # print every 250 mini-batches
            print('[{}, {}] loss: {:.3f}'.format(epoch + 1, i + 1,
             ↪ running_loss / 250))
            running_loss_list.append(running_loss)
            running_loss = 0.0
    epoch_loss = running_loss / total_batches
    average_loss_per_epoch.append(epoch_loss)
    print(f'Epoch {epoch+1}, Loss: {epoch_loss:.4f}')

print('Training Complete')
PATH = './net.pth'
torch.save(net.state_dict(), PATH)

# complete the code to plot the running loss per 250 mini batches curve

def plot_loss_curve(running_loss_list):
  ## complete code
    plt.plot(running_loss_list)
    plt.ylabel('Loss')
    plt.xlabel('Every 250 mini-batches')
    plt.title('Training Loss')
    plt.show()

plot_loss_curve(running_loss_list)
```

(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

```
# SLightly Modified Code Here - SL
num_epochs = 20
running_loss_list = []  # list to store running loss in the code below
average_loss_per_epoch = []

for epoch in range(num_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    total_batches = 0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data[0].cuda(), data[1].cuda()  # Ensure data
        ↪ is on GPU

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        total_batches += 1
        if i % 250 == 249:    # print every 250 mini-batches
            print(f'[{epoch + 1}, {i + 1}] loss: {running_loss /
            ↪ 250:.3f}')
            running_loss_list.append(running_loss / 250)  # Corrected to
            ↪ store the average
            running_loss = 0.0

    epoch_loss = running_loss / total_batches if total_batches else 0
    average_loss_per_epoch.append(epoch_loss)
    print(f'Epoch {epoch+1}, Loss: {epoch_loss:.4f}')

print('Training Complete')
PATH = './net.pth'
torch.save(net.state_dict(), PATH)

plot_loss_curve(running_loss_list)
```

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

Resetting the parameter gradients to zero at the beginning of each training iteration is a critical step in neural network training. This process is necessary due to PyTorch's default behavior of accumulating gradients on subsequent backward passes. Without resetting, gradients from the current batch of data would be mixed with gradients from previous batches, leading to incorrect parameter updates. The primary reasons for this procedure are to prevent accumulation from previous iterations, ensure correct gradient computation which will lead to more accurate parameter updates.

To reset gradients in PyTorch, the `.zero_grad()` method is used on the optimizer:

```
optimizer.zero_grad()
```

This method clears the gradients of all optimized parameters, setting the stage for a clean gradient computation for the subsequent training batch.

### 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.

```
### Accuracy on whole data 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  = acc = 100 * correct / total
print('Accuracy of the network on the 10000 test images: %d %%' % (acc))
```

(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

```
from tqdm import tqdm
def train(train_loader=train_loader, num_epochs=2, use_gpu=False,
 ↪  lr=0.001, momentum=0.9, model_save_path='./net.pth'):
  """
  INPUTS
  num_epochs: number of training epochs
  use_gpu: False by default. If true, load the model and data to GPU
   ↪  for training
  lr: learning rate for SGD optimizer
  momentum: momentum for SGD optimizer
  model_save_path: save path for the trained model

  OUTPUTS
  returns running_loss_list: which stores the loss averaged over a
   ↪  minibatch of size 250

  Author: Simon Lee
  """
  net = Net()
  if use_gpu:
      net.cuda()

  criterion = nn.CrossEntropyLoss()
  optimizer = optim.SGD(net.parameters(), lr=lr, momentum=momentum)

  running_loss_list = []
  for epoch in tqdm(range(num_epochs)):
      running_loss = 0.0
      for i, data in enumerate(train_loader, 0):
          inputs, labels = data
          if use_gpu:
              inputs, labels = inputs.cuda(), labels.cuda()

          optimizer.zero_grad()

          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()

          running_loss += loss.item()
          if i % 250 == 249:
```

```
                running_loss_list.append(running_loss / 250)
                running_loss = 0.0
        print(f'Epoch {epoch+1} completed')

    torch.save(net.state_dict(), model_save_path)
    return running_loss_list

def test(test_loader=test_loader, model_path='./net.pth'):
    """
    Author: Simon Lee
    """

    ### complete the code to compute accuracy and store it as the
     ↪  variable acc
    net = Net()
    net.load_state_dict(torch.load(model_path))
    net.cuda() if next(net.parameters()).is_cuda else net

    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            if next(net.parameters()).is_cuda:
                images, labels = images.cuda(), labels.cuda()
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    acc = 100 * correct / total
    print('Accuracy of the network on the 10000 test images: %d %%' %
     ↪  (acc))
    return acc

# main
batch_size = 4 ## set the batch size value
train_loader = torch.utils.data.DataLoader(train_data,
 ↪  batch_size=batch_size, shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_data,
 ↪  batch_size=batch_size, shuffle=False, num_workers=2)

# BATCH 4 EPOCH 5
```

```
loss = train(train_loader=train_loader, num_epochs=5, use_gpu=True,
 ↪  lr=0.001, momentum=0.9, model_save_path='./net.pth')
_ = test(test_loader=test_loader, model_path='./net.pth')
plot_loss_curve(running_loss_list)
# BATCH 4 EPOCH 20
loss = train(train_loader=train_loader, num_epochs=20, use_gpu=True,
 ↪  lr=0.001, momentum=0.9, model_save_path='./net.pth')
_ = test(test_loader=test_loader, model_path='./net.pth')
plot_loss_curve(running_loss_list)

batch_size = 16 ## set the batch size value
train_loader = torch.utils.data.DataLoader(train_data,
 ↪  batch_size=batch_size, shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_data,
 ↪  batch_size=batch_size, shuffle=False, num_workers=2)

# BATCH 16 EPOCH 5
loss = train(train_loader=train_loader, num_epochs=5, use_gpu=True,
 ↪  lr=0.001, momentum=0.9, model_save_path='./net.pth')
_ = test(test_loader=test_loader, model_path='./net.pth')
plot_loss_curve(running_loss_list)
# BATCH 16 EPOCH 20
loss = train(train_loader=train_loader, num_epochs=20, use_gpu=True,
 ↪  lr=0.001, momentum=0.9, model_save_path='./net.pth')
_ = test(test_loader=test_loader, model_path='./net.pth')
plot_loss_curve(running_loss_list)
```

(iii) Explain your observations in (ii)

More epochs is better and a larger batch size performs the best on the CIFAR 10 dataset. However in general the model doesn't seem to be optimized for the best performance because even with 10 classes it seems to do pretty bad with a max of 65% accuracy. However more epochs may also result in overfitting.

# 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)** Why batch normalization acts as a regularizer.

As a regularizer, Batch Normlaization (BN) reduces internal covariate shift and introduces noise through mini-batch statistics, which enhances model generalization by preventing over-reliance on specific features or training paths which mimic the dropout feature. This noise and variability, derived from computing mean and variance on mini-batches, inadvertently add a regularization effect, similar to introducing randomness in the training process.

**(ii)** Difference in using batch normalization at training vs inference (testing) time.

However, at inference time, it switches to using these pre-computed population statistics to ensure model outputs remain consistent and deterministic for any given input, diverging from the mini-batch based calculation to achieve stable and reliable predictions. This dual behavior optimizes model performance by leveraging batch-wise variability for regularization during training while maintaining consistency during inference

### 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 $C = 64, H = 32, W = 32$

For a convolutional layer with parameters $C_{in} = 32$, $C_{out} = 64$, and kernel size $k = 3$, applied to an input of $C = 32, H = 64, W = 64$ we can use the following to calculate the next two paerts:

$$H_{out} = \frac{H - k + 2P}{S} + 1$$
$$W_{out} = \frac{W - k + 2P}{S} + 1$$

(i) Without padding and with a stride of 1, the output size is calculated as $H_{out} = W_{out} = \frac{64-3+0}{1} + 1 = 62$, resulting in an output dimension of $C = 64, H = 62, W = 62$.

(ii) To achieve an output size of $C = 64, H = 32, W = 32$, a stride ($S$) of 2 is required. For padding ($P$), solving $\frac{64-3+2P}{2} + 1 = 32$ yields $P = 0.5$, which is practically rounded to 1. Thus, a stride of 2 and padding of 1 are needed.

### 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 $\lambda$ 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.

$L2$ regularization introduces a penalty term, $\lambda w^T w$, to the original loss function, $L(y,\hat{y})$, resulting in a modified loss function $L'(y,\hat{y}) = L(y,\hat{y}) + \lambda w^T w$. This approach penalizes the square of the weights, leading to smaller weight values. In the gradient descent update formula, the inclusion of $L2$ regularization adds a term $2\lambda w$ to the derivative of the loss with respect to the weights, modifying the update rule to $w = w - \alpha(\frac{\partial L(y,\hat{y})}{\partial w} + 2\lambda w)$. This modification causes the weights to shrink slightly towards zero with each update—a process known as weight decay—effectively simplifying the model and preventing overfitting by discouraging the use of large weights, thereby promoting a more generalized model.

### 3.4 Why CNNs? (3 points)

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

1. CNNs require fewer parameters compared to fully connected networks due to shared weights and local connectivity, making them more efficient for image data.

2. CNNs can capture spatial hierarchies in images through the use of convolutional and pooling layers, allowing them to recognize patterns and objects at various scales and orientations.

# 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?

The overall GAN loss function is:

$$L = \min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$$

For an optimally trained GAN:

(a) Ideal discriminator behavior:

$$D(x) = 1, D(G(z)) = 0$$

(b) Overall loss function value:

$$L = -\log(4)$$

## 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.*

Given the GAN architecture and activation functions, the gradients of the GAN loss function with respect to the weight matrices $\mathbf{W_g}$ and $\mathbf{W_d}$ involve applying the chain rule to the composition of functions that define the generator and discriminator outputs.

Generator Output:

$$\mathbf{y} = ReLU(\mathbf{W_g x})$$

Discriminator Output:

$$b = Sigmoid(\mathbf{W_d z})$$

, where $\mathbf{z} = \mathbf{y}$ for the generated data.

Derivative of Sigmoid:

$$Sigmoid'(x) = Sigmoid(x)(1 - Sigmoid(x))$$

Derivative of ReLU:

$$ReLU'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

Gradient with respect to $\mathbf{W_d}$:

$$\frac{\partial L}{\partial \mathbf{W_d}} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial \mathbf{W_d}} = \frac{\partial L}{\partial b} \cdot Sigmoid'(b) \cdot \mathbf{z}^T$$

Gradient with respect to $\mathbf{W_g}$:

$$\frac{\partial L}{\partial \mathbf{W_g}} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W_g}}$$

Given $b = Sigmoid(\mathbf{W_d y})$, the derivative $\frac{\partial b}{\partial \mathbf{y}}$ is:

$$\frac{\partial b}{\partial \mathbf{y}} = Sigmoid'(b) \cdot \mathbf{W_d}$$

And given $\mathbf{y} = ReLU(\mathbf{W_g x})$, the derivative $\frac{\partial \mathbf{y}}{\partial \mathbf{W_g}}$ is:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{W_g}} = ReLU'(\mathbf{W_g x}) \cdot \mathbf{x}^T$$

The whole expression involdves computing the derivatives of the loss funciton and applying the chain rule.

### 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 a Generative Adversarial Network (GAN) with inputs from varying probability distributions, rather than a consistent one like the normal distribution, introduces significant challenges that lead to training instability and impede the model's efficiency. In the standard GAN framework, the generator (G) aims to replicate the actual data distribution from a fixed noise distribution ($p_z(z)$), while the discriminator (D) seeks to distinguish between real and generated data. However, employing variable noise distributions complicates G's adaptation process, resulting in erratic training behavior due to the difficulty in adjusting to continuously shifting distributions. Furthermore, these fluctuations disrupt G's ability to establish a stable mapping from noise to data, further hindering its learning efficiency.