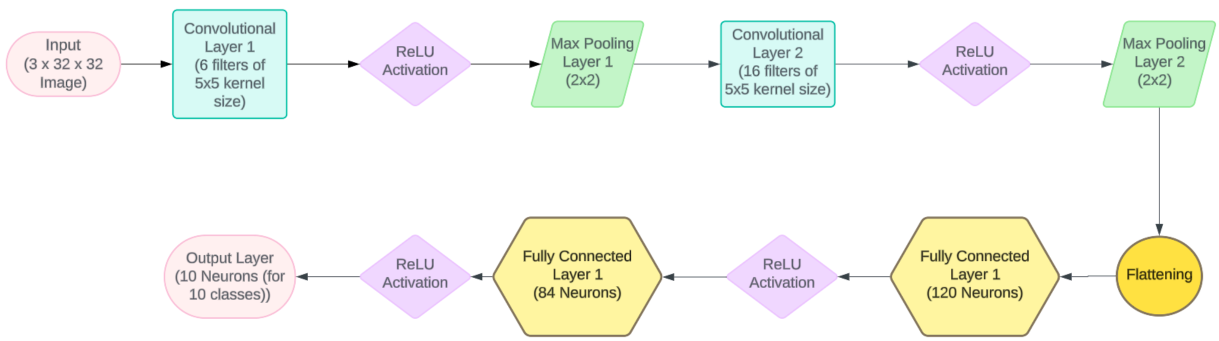
<u>MSDA-3100-01 Applied Deep Learning Assignment 2</u>

Problem 1

1) Block Diagram of the Implemented Network Architecture



Description:

1. **Input Layer**: Accepts a 3-channel image of size 32x32 pixels.
2. **Convolutional Layer 1**: Applies 6 convolutional filters of size 5x5, resulting in a 6-channel feature map.
3. **Activation Layer 1**: Uses ReLU (Rectified Linear Unit) activation to introduce non-linearity.
4. **Max Pooling Layer 1**: Applies a 2x2 max pooling operation to reduce the spatial dimension.
5. **Convolutional Layer 2**: Applies 16 convolutional filters of size 5x5, resulting in a 16-channel feature map.
6. **Activation Layer 2**: Uses ReLU activation.
7. **Max Pooling Layer 2**: Applies a 2x2 max pooling operation to further reduce the spatial dimension.
8. **Flattening**: Converts the 16-channel output into a 1D vector (size: 16 * 5 * 5).
9. **Fully Connected Layer 1**: 120 neurons.
10. **Activation Layer 3**: Uses ReLU activation.
11. **Fully Connected Layer 2**: 84 neurons.
12. **Activation Layer 4**: Uses ReLU activation.
13. **Output Layer**: 10 neurons representing each class (e.g., plane, car, bird, etc.).

2) Dimensions of the Generated Features After the First Convolutional Layer (Before Pooling)

The dimensions of the generated features after the first convolutional layer (before the pooling layer) can be calculated as follows:

Formula:

For a convolutional layer, the output dimensions can be calculated using the formula:

Output Dimension = ((Input Dimension - Kernel Size + 2 x Padding) / Stride) - 1

Values from the Code:

- **Input Dimensions**: 32 x 32 (height x width)
- **Kernel Size**: 5 (5x5 kernel)
- **Padding**: 0 (no padding applied in the code)
- **Stride**: 1 (default stride value)

Calculation:

1. **Output Height and Width**:

   Output Height = Output Width = (32 - 5 + 0) / 1) + 1 = 28

2. Number of Output Channels: 6 (As defined by the first convolutional layer: **`self.conv1 = nn.Conv2d(3, 6, 5)`**)

Final Dimensions:

The dimensions of the generated feature map after the first convolutional layer are:

[6, 28, 28]

This indicates:

- **6 output channels,**
- **Each of size 28 x 28 (height x width).**

```python
# Define a function to print the feature map sizes
def print_feature_map_sizes():
    # Create a dummy input of size (1, 3, 32, 32) as CIFAR-10 images have shape 32x32x3
    dummy_input = torch.randn(1, 3, 32, 32)

    # Pass through the first convolutional layer
    out_conv1 = net.conv1(dummy_input)
    print(f"Dimensions after first convolutional layer (before pooling): {out_conv1.shape}")

    # Pass through the first pooling layer
    out_pool1 = net.pool(out_conv1)
    print(f"Dimensions after first pooling layer: {out_pool1.shape}")

    # Pass through the second convolutional layer
    out_conv2 = net.conv2(out_pool1)
    print(f"Dimensions after second convolutional layer (before second pooling): {out_conv2.shape}")

    # Pass through the second pooling layer
    out_pool2 = net.pool(out_conv2)
    print(f"Dimensions after second pooling layer: {out_pool2.shape}")

# Run the function to print dimensions
print_feature_map_sizes()
```

```
Dimensions after first convolutional layer (before pooling): torch.Size([1, 6, 28, 28])
Dimensions after first pooling layer: torch.Size([1, 6, 14, 14])
Dimensions after second convolutional layer (before second pooling): torch.Size([1, 16, 10, 10])
Dimensions after second pooling layer: torch.Size([1, 16, 5, 5])
```

3) Dimensions of the Generated Features After the First Convolutional Layer (After Pooling)

After the first convolutional layer, the feature dimensions are passed through a max pooling layer (`self.pool = nn.MaxPool2d(2, 2)`). The max pooling layer has the following properties:

- **Kernel Size:** 2
- **Stride:** 2

This means that the pooling operation will reduce the height and width of the feature maps by a factor of 2.

Input Dimensions to the Pooling Layer:

The feature dimensions after the first convolutional layer (before pooling) are:

- **Channels: 6**
- **Height: 28**
- **Width: 28**

Formula for Pooling:

The output dimensions after the pooling operation can be calculated using the formula:

Output Dimension = Input Dimension / Pooling Stride

Calculation:

1. **Output Height**:
    a. Output Height = 28 / 2 = 14
2. **Output Width**
    a. Output Width = 28 / 2 = 14
3. **Number of Channels**: Remains the same as before pooling, i.e., 6 channels.

Final Dimensions after Pooling:

The dimensions of the feature maps after the pooling layer are:

[6, 14, 14]

This indicates:

- 6 output channels,
- Each of size 14 x 14 (height x width).

```python
# Define a function to print the feature map sizes
def print_feature_map_sizes():
    # Create a dummy input of size (1, 3, 32, 32) as CIFAR-10 images have shape 32x32x3
    dummy_input = torch.randn(1, 3, 32, 32)

    # Pass through the first convolutional layer
    out_conv1 = net.conv1(dummy_input)
    print(f"Dimensions after first convolutional layer (before pooling): {out_conv1.shape}")

    # Pass through the first pooling layer
    out_pool1 = net.pool(out_conv1)
    print(f"Dimensions after first pooling layer: {out_pool1.shape}")

    # Pass through the second convolutional layer
    out_conv2 = net.conv2(out_pool1)
    print(f"Dimensions after second convolutional layer (before second pooling): {out_conv2.shape}")

    # Pass through the second pooling layer
    out_pool2 = net.pool(out_conv2)
    print(f"Dimensions after second pooling layer: {out_pool2.shape}")

# Run the function to print dimensions
print_feature_map_sizes()
```

```
Dimensions after first convolutional layer (before pooling): torch.Size([1, 6, 28, 28])
Dimensions after first pooling layer: torch.Size([1, 6, 14, 14])
Dimensions after second convolutional layer (before second pooling): torch.Size([1, 16, 10, 10])
Dimensions after second pooling layer: torch.Size([1, 16, 5, 5])
```

4) Dimensions of the Generated Features After the Second Convolutional Layer (Before the First Fully Connected Layer)

The feature dimensions after the second convolutional layer (before the first fully connected layer) can be calculated as follows:

Step 1: Dimensions After the Second Convolutional Layer

The input dimensions to the second convolutional layer are the output dimensions from the first pooling layer:

- **Input Dimensions to Second Convolutional Layer: [6, 14, 14]**
- **Number of Output Channels: 16 (as defined by `self.conv2 = nn.Conv2d(6, 16, 5)`)**
- **Kernel Size: 5 (5x5 filter)**
- **Padding: 0 (no padding is used)**
- **Stride: 1 (default stride value)**

Applying the Convolutional Formula:

The formula to calculate the output dimensions is:

Output DImension = ((Input Dimension - Kernel Size + 2 x Padding)/ Stride) + 1

Plugging in the values:

1. **Output Height and Width**:

   Output Height = Output Width = ((14 - 5 +0) / 1) + 1

2. **Number of Output Channels**: 16 (as defined in the second convolutional layer)

Step 2: Dimensions After the Second Max Pooling Layer

The output of the second convolutional layer is passed through a max pooling layer (**`self.pool`**) with the following properties:

- **Pooling Kernel Size**: 2
- **Pooling Stride**: 2

The formula for calculating the dimensions after pooling is:

Output Dimension = Input Dimension / Pooling Stride

3. **Output Height**:

Output Height = 10 / 2 = 5

4. **Output Width**

Output Width = 10 / 2 = 5

5. **Number of Channels**: 16 (remains the same as before pooling)

Final Dimensions Before the Fully Connected Layer:

The final dimensions of the feature map before entering the first fully connected layer are:

[16, 5, 5]

Step 3: Flattening the Dimensions

Before passing to the fully connected layer, the feature maps are flattened into a 1D vector:

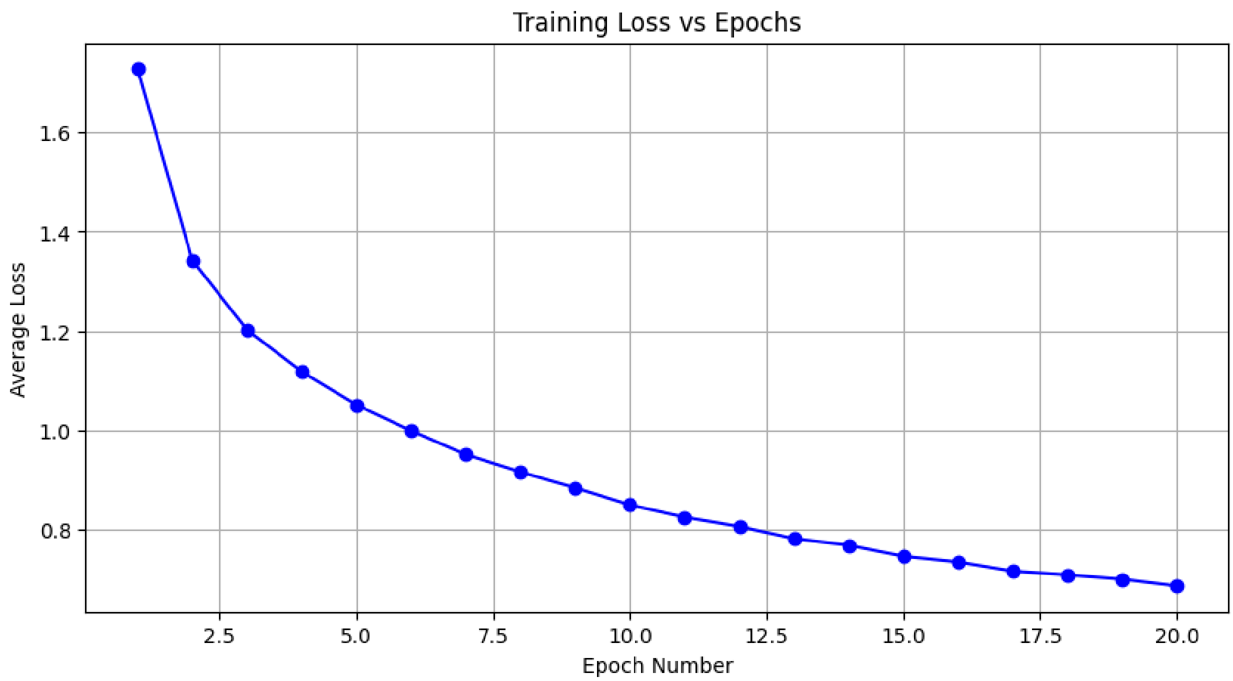
- Flattened Size: 16×5×5=40016 \times 5 \times 5 = 40016×5×5=400

So, the input to the first fully connected layer is a 1D tensor of size **400**.

```python
# Define a function to print the feature map sizes
def print_feature_map_sizes():
    # Create a dummy input of size (1, 3, 32, 32) as CIFAR-10 images have shape 32x32x3
    dummy_input = torch.randn(1, 3, 32, 32)

    # Pass through the first convolutional layer
    out_conv1 = net.conv1(dummy_input)
    print(f"Dimensions after first convolutional layer (before pooling): {out_conv1.shape}")

    # Pass through the first pooling layer
    out_pool1 = net.pool(out_conv1)
    print(f"Dimensions after first pooling layer: {out_pool1.shape}")

    # Pass through the second convolutional layer
    out_conv2 = net.conv2(out_pool1)
    print(f"Dimensions after second convolutional layer (before second pooling): {out_conv2.shape}")

    # Pass through the second pooling layer
    out_pool2 = net.pool(out_conv2)
    print(f"Dimensions after second pooling layer: {out_pool2.shape}")

# Run the function to print dimensions
print_feature_map_sizes()

```

```
Dimensions after first convolutional layer (before pooling): torch.Size([1, 6, 28, 28])
Dimensions after first pooling layer: torch.Size([1, 6, 14, 14])
Dimensions after second convolutional layer (before second pooling): torch.Size([1, 16, 10, 10])
Dimensions after second pooling layer: torch.Size([1, 16, 5, 5])
```

5) Changing the Number of Epochs to 20 and Plotting the Average Loss for Each Epoch



6) Total Average Training Accuracy after 20 epochs

Total Average Training Accuracy after 20 epochs: 78.88%

```python
# Calculating the accuracy of the trained model on the entire training dataset
total_correct_train = 0
total_train_samples = 0

# Putting the network in evaluation mode (this disables dropout and batch normalization)
net.eval()

# Ignoring tracking gradients
with torch.no_grad():
    for data in trainloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = net(inputs)

        # Get predictions
        _, predicted = torch.max(outputs, 1)

        # Count the correct predictions
        total_train_samples += labels.size(0)
        total_correct_train += (predicted == labels).sum().item()

# Calculating the total average training accuracy
total_train_accuracy = 100 * total_correct_train / total_train_samples
print(f'Total Average Training Accuracy after 20 epochs: {total_train_accuracy:.2f}%')
```

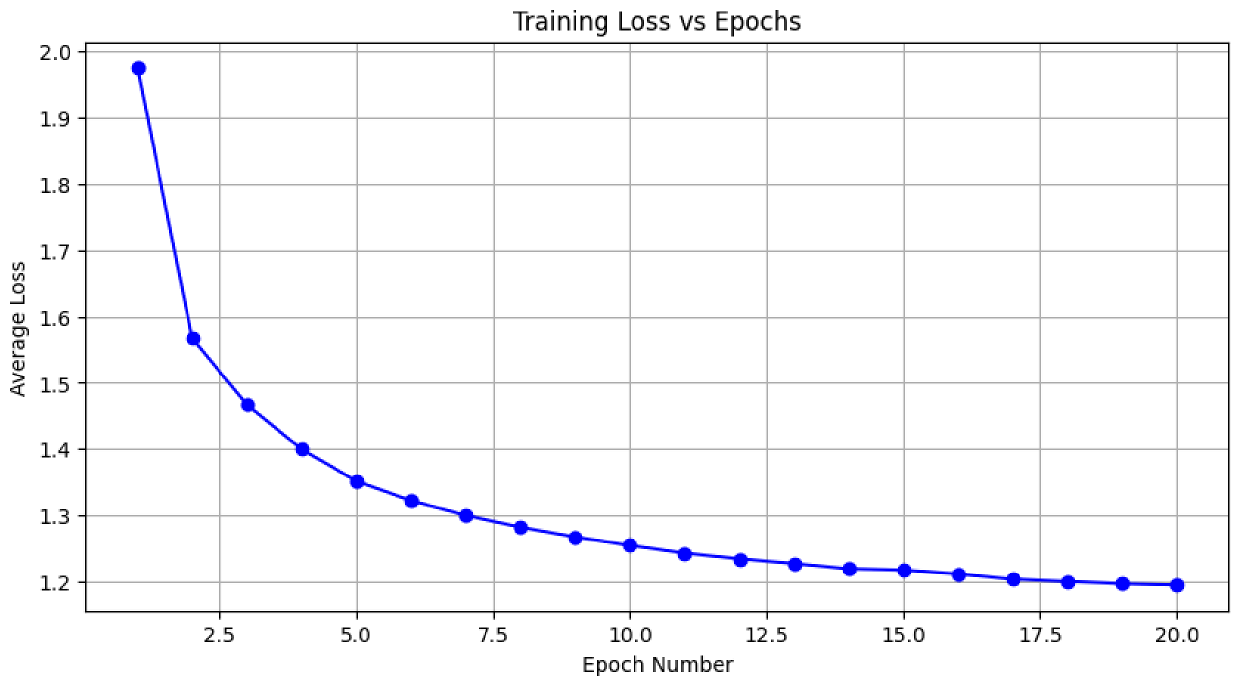
7) Total Average Testing Accuracy after 20 epochs

Total Average Testing Accuracy after 20 epochs: 60.05%

```python
# Calculating the accuracy of the trained model on the entire testing dataset
total_correct_test = 0
total_test_samples = 0

# Putting the network in evaluation mode (this disables dropout and batch normalization)
net.eval()

# Ignoring tracking gradients
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass through the network
        outputs = net(inputs)

        # Get predictions from the maximum value
        _, predicted = torch.max(outputs, 1)

        # Count the total number of correct predictions
        total_test_samples += labels.size(0)
        total_correct_test += (predicted == labels).sum().item()

# Calculating the total average testing accuracy
total_test_accuracy = 100 * total_correct_test / total_test_samples
print(f'Total Average Testing Accuracy after 20 epochs: {total_test_accuracy:.2f}%')
```

Problem 2

1) Calculating the average loss of all batches for every epoch and plotting the average loss on the y-axis vs the epoch number on the x-axis.



2) Total average accuracy of all classes of the training dataset after epoch 20.

Total Average Training Accuracy after 20 epochs: 58.53%

3) Total average accuracy of all classes of the testing dataset after epoch 20.

Total Average Testing Accuracy after 20 epochs: 54.80%

4) Problem 1 and 2 performance comparison

Based on the outputs provided in the notebook, here's a comparison of the performance between **Problem 1** and **Problem 2**:

1. **Problem 1 (Original Network)**:

    - **Training Accuracy**: 78.88%
    - **Testing Accuracy**: 60.05%
2. **Problem 2 (Modified Network with Additional Convolutional Layer)**:
    - **Training Accuracy**: 58.53%
    - **Testing Accuracy**: 54.80%

Which Network is Better?

- **Problem 1's network** is better in terms of both training and testing accuracy. It achieved a significantly higher training accuracy of **78.88%** compared to **58.53%** for Problem 2, and a testing accuracy of **60.05%** compared to **54.80%** for Problem 2.

Conclusion:

The **original network from Problem 1** outperformed the modified network from Problem 2. Thus, **Problem 1's network is better in performance**.

Code Description and Procedure

The project aimed to compare the performance of two Convolutional Neural Network (CNN) architectures using the CIFAR-10 dataset, which consists of 60,000 color images divided into 10 classes. Two different CNN architectures were evaluated: the original network and a modified network with an additional convolutional layer. The project involved several key steps:

1. **Data Preparation**: The CIFAR-10 dataset was used, which contains 32x32 pixel images. These images were preprocessed and split into training and testing datasets.
2. **Implementation of the CNN Architectures**:
    - **Original Network (Problem 1)**:
        - **Input Layer**: Accepts a 3-channel image of size 32x32 pixels.
        - **Convolutional Layer 1**: Applies 6 filters of size 5x5, resulting in a 6-channel feature map.
        - **ReLU Activation Layer**: Introduces non-linearity.
        - **Max Pooling Layer**: Reduces the spatial dimension by half.
        - **Convolutional Layer 2**: Applies 16 filters of size 5x5, producing a 16-channel feature map.
        - **ReLU Activation Layer**: Adds non-linearity.
        - **Max Pooling Layer**: Further reduces the spatial dimension.
        - **Flattening**: Converts the feature map to a 1D vector.
        - **Fully Connected Layers**: Two layers with 120 and 84 neurons respectively, each followed by ReLU activation.
        - **Output Layer**: 10 neurons for the 10 classes.
    - **Modified Network (Problem 2)**: The original network was modified by adding an additional convolutional layer with 32 filters of size 5x5 before the final max pooling layer.

3. **Training and Evaluation**:
   - Both architectures were trained for 20 epochs.
   - The average loss and accuracy for each epoch were recorded for training and testing datasets.
   - Training and testing accuracy for each model was compared.

Goal of the Project

The goal was to evaluate the impact of an additional convolutional layer on the performance of a CNN architecture for image classification on the CIFAR-10 dataset. The project sought to determine whether a deeper network with more convolutional layers would yield improved performance compared to the original shallower network.

Results

- **Original Network (Problem 1)**:
  - Total Average Training Accuracy: **78.88%**
  - Total Average Testing Accuracy: **60.05%**
- **Modified Network with Additional Convolutional Layer (Problem 2)**:
  - Total Average Training Accuracy: **58.53%**
  - Total Average Testing Accuracy: **54.80%**

Final Conclusion and Personal Comments

The original network performed significantly better than the modified network, both in terms of training and testing accuracy. This result suggests that simply adding more convolutional layers does not necessarily improve the performance of a CNN on a relatively simple dataset like CIFAR-10. The added layer might have led to overfitting or increased model complexity, making it harder for the model to generalize well on unseen data.

Personal Comments: The project highlights the importance of balancing model complexity with dataset characteristics. While deeper networks are often preferred for more complex tasks, they may not always be the best choice for smaller datasets. In this case, the additional convolutional layer likely increased the number of parameters without providing significant benefits, leading to lower overall performance. Future work could explore other modifications, such as regularization techniques or different activation functions, to better leverage the increased depth of the network.

Code Snippets

Problem 1

```
import torch
import torchvision
import torchvision.transforms as transforms

# These two lines can be used in case you have multiple versions of the
OpenMP library
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'


########################################################################
# How do we run these neural networks on the GPU?
#
# Training on GPU
# ----------------
# Just like how you transfer a Tensor onto the GPU, you transfer the
neural
# net onto the GPU.
#
# Let's first define our device as the first visible cuda device if we
have
# CUDA available:

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Assuming that we are on a CUDA machine, this should print a CUDA device:

print(device)

########################################################################
# The output of torchvision datasets are PILImage images of range [0, 1].
# We transform them to Tensors of normalized range [-1, 1].

########################################################################
# .. note::
#     If running on Windows and you get a BrokenPipeError, try setting
#     the num_worker of torch.utils.data.DataLoader() to 0.

transform = transforms.Compose(
```

```
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4      #You can change the batch size if you receive an out of
memory error

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=0)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

########################################################################
# Let us show some of the training images, for fun.

import matplotlib.pyplot as plt
import numpy as np

# functions to unnormalize and show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)
```

```
# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))


########################################################################
# 2. Define a Convolutional Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Copy the neural network from the Neural Networks section before and
modify it to
# take 3-channel images (instead of 1-channel images as it was defined).

import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

########################################################################
```

```
# 3. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Let's use a Classification Cross-Entropy loss and SGD with momentum.

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

########################################################################
# 4. Train the network
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to
the
# network and optimize.

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # 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()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss /
2000:.3f}')
            running_loss = 0.0
```

```
print('Finished Training')

########################################################################
# Let's quickly save our trained model:

PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

########################################################################
# 5. Test the network on the test data
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#
# We have trained the network for 2 passes over the training dataset.
# But we need to check if the network has learnt anything at all.
#
# We will check this by predicting the class label that the neural network
# outputs, and checking it against the ground-truth. If the prediction is
# correct, we add the sample to the list of correct predictions.
#
# Okay, first step. Let us display an image from the test set to get
familiar.

dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in
range(4)))

########################################################################
# Next, let's load back in our saved model (note: saving and re-loading
the model
# wasn't necessary here, we only did it to illustrate how to do so):

net = Net()
net.load_state_dict(torch.load(PATH))

########################################################################
```

```python
# Okay, now let us see what the neural network thinks these examples above
are:

outputs = net(images)

########################################################################
# The outputs are energies for the 10 classes.
# The higher the energy for a class, the more the network
# thinks that the image is of the particular class.
# So, let's get the index of the highest energy:
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))

########################################################################
# The results seem pretty good.
#
# Let us look at how the network performs on the whole dataset.

correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for
our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as
prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct
// total} %')

########################################################################
```

```
# That looks way better than chance, which is 10% accuracy (randomly
picking
# a class out of 10 classes).
# Seems like the network has learned something.
#
# Hmmm, what are the classes that performed well, and the classes that did
# not perform well:

# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

Continuing Main Code with 20 Epochs

```
import matplotlib.pyplot as plt

# Changing the number of epochs to 20
num_epochs = 20

# Storing the average loss for each epoch
```

```
epoch_losses = []

# Training the network with 20 epochs
for epoch in range(num_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    total_batches = 0

    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

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

        # accumulate loss
        running_loss += loss.item()
        total_batches += 1

    # Calculating the average loss for the epoch
    avg_loss = running_loss / total_batches
    epoch_losses.append(avg_loss)
    print(f'Epoch [{epoch + 1}/{num_epochs}], Average Loss:
{avg_loss:.4f}')

# Plotting the average loss for each epoch
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), epoch_losses, marker='o')
plt.title('Average Loss per Epoch')
plt.xlabel('Epoch Number')
plt.ylabel('Average Loss')
plt.grid()
plt.show()
```

Modifying Main Code with 20 Epochs and Seeing the Training Rate for Better Results

```
import torch
import torchvision
import torchvision.transforms as transforms
import os
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# Setting up environment to avoid OpenMP library issues
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'

# Defining device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(f'Running on device: {device}')

# Transforming for normalization and conversion to tensor
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Loading CIFAR10 dataset
batch_size = 4
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=0)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)

classes = ('plane', 'car', 'bird', 'cat',
```

```python
        'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# Defining CNN model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # Flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net().to(device)

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

# Training the network with 20 epochs
epochs = 20
average_loss_per_epoch = []

for epoch in range(epochs):  # Training loop for 20 epochs
    running_loss = 0.0
    epoch_loss = 0.0
    total_batches = 0

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        # Zero the parameter gradients
        optimizer.zero_grad()

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

        # Track loss
        running_loss += loss.item()
        epoch_loss += loss.item()
        total_batches += 1

        # Print statistics every 2000 mini-batches
        if i % 2000 == 1999:
            print(f'[Epoch {epoch + 1}, Batch {i + 1}] loss: {running_loss
/ 2000:.3f}')
            running_loss = 0.0

    # Calculate average loss for the epoch
    average_epoch_loss = epoch_loss / total_batches
    average_loss_per_epoch.append(average_epoch_loss)
    print(f'Epoch {epoch + 1} completed with Average Loss:
{average_epoch_loss:.4f}')

print('Finished Training')

# Saving the trained model
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

# Plotting the average loss for each epoch
plt.figure(figsize=(10, 5))
plt.plot(range(1, epochs + 1), average_loss_per_epoch, marker='o',
linestyle='-', color='b')
plt.title('Training Loss vs Epochs')
plt.xlabel('Epoch Number')
plt.ylabel('Average Loss')
```

```
plt.grid()
plt.show()
```

Total Average Training Accuracy after 20 epochs

```
# Calculating the accuracy of the trained model on the entire training
dataset
total_correct_train = 0
total_train_samples = 0

# Putting the network in evaluation mode (this disables dropout and batch
normalization)
net.eval()

# Ignoring tracking gradients
with torch.no_grad():
    for data in trainloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = net(inputs)

        # Get predictions
        _, predicted = torch.max(outputs, 1)

        # Count the correct predictions
        total_train_samples += labels.size(0)
        total_correct_train += (predicted == labels).sum().item()

# Calculating the total average training accuracy
total_train_accuracy = 100 * total_correct_train / total_train_samples
print(f'Total Average Training Accuracy after 20 epochs:
{total_train_accuracy:.2f}%')
```

Total Average Testing Accuracy after 20 epochs

```
# Calculating the accuracy of the trained model on the entire testing
dataset
```

```python
total_correct_test = 0
total_test_samples = 0

# Putting the network in evaluation mode (this disables dropout and batch
normalization)
net.eval()

# Ignoring tracking gradients
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass through the network
        outputs = net(inputs)

        # Get predictions from the maximum value
        _, predicted = torch.max(outputs, 1)

        # Count the total number of correct predictions
        total_test_samples += labels.size(0)
        total_correct_test += (predicted == labels).sum().item()

# Calculating the total average testing accuracy
total_test_accuracy = 100 * total_correct_test / total_test_samples
print(f'Total Average Testing Accuracy after 20 epochs:
{total_test_accuracy:.2f}%')
```

Problem 2

```python
# Setting up environment to avoid OpenMP library issues
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'

# Defining device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(f'Running on device: {device}')

# Transforming for normalization and conversion to tensor
transform = transforms.Compose(
```

```
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Loading CIFAR10 dataset
batch_size = 4
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=0)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# Defining a Modified CNN model with an extra convolutional layer
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)       # First convolutional layer
        self.pool1 = nn.MaxPool2d(2, 2)        # First max pooling layer
        self.conv2 = nn.Conv2d(6, 16, 5)       # Second convolutional
layer
        self.pool2 = nn.MaxPool2d(2, 2)        # Second max pooling layer
        self.conv3 = nn.Conv2d(16, 10, 3)      # Third convolutional layer
(new layer)
        self.pool3 = nn.MaxPool2d(2, 2)        # Third max pooling layer
(new)

        # Adjust the dimensions of the linear layers accordingly
        self.fc1 = nn.Linear(10 * 1 * 1, 120)  # Fully connected layer
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
```

```python
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = torch.flatten(x, 1)  # Flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate the modified network
net = Net().to(device)

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

# Training the network with 20 epochs and tracking the average loss for
each epoch
epochs = 20
average_loss_per_epoch = []

for epoch in range(epochs):  # Training loop for 20 epochs
    running_loss = 0.0
    epoch_loss = 0.0
    total_batches = 0

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

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

        # Track loss
        running_loss += loss.item()
```

```
        epoch_loss += loss.item()
        total_batches += 1

        # Print statistics every 2000 mini-batches
        if i % 2000 == 1999:
            print(f'[Epoch {epoch + 1}, Batch {i + 1}] loss: {running_loss
/ 2000:.3f}')
            running_loss = 0.0

    # Calculating average loss for the epoch
    average_epoch_loss = epoch_loss / total_batches
    average_loss_per_epoch.append(average_epoch_loss)
    print(f'Epoch {epoch + 1} completed with Average Loss:
{average_epoch_loss:.4f}')

print('Finished Training')

# Saving the trained model
PATH = './modified_cifar_net.pth'
torch.save(net.state_dict(), PATH)

# 1. Plotting the average loss for each epoch
plt.figure(figsize=(10, 5))
plt.plot(range(1, epochs + 1), average_loss_per_epoch, marker='o',
linestyle='-', color='b')
plt.title('Training Loss vs Epochs')
plt.xlabel('Epoch Number')
plt.ylabel('Average Loss')
plt.grid()
plt.show()

# 2. Calculating the average accuracy of the training dataset
total_correct_train = 0
total_train_samples = 0
net.eval()

with torch.no_grad():
    for data in trainloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
```

```python
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total_train_samples += labels.size(0)
        total_correct_train += (predicted == labels).sum().item()

total_train_accuracy = 100 * total_correct_train / total_train_samples
print(f'Total Average Training Accuracy after {epochs} epochs:
{total_train_accuracy:.2f}%')

# 3. Calculating the average accuracy of the testing dataset
total_correct_test = 0
total_test_samples = 0

with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total_test_samples += labels.size(0)
        total_correct_test += (predicted == labels).sum().item()

total_test_accuracy = 100 * total_correct_test / total_test_samples
print(f'Total Average Testing Accuracy after {epochs} epochs:
{total_test_accuracy:.2f}%')
```