EE449 HW1

# **Basic Concepts**

## **Which Function?**

An ANNs classifier trained with cross-entropy loss approximates the function of conditional probability distribution, which approximates the mapping from input to output probabilities.

The loss is defined with a scoring system that compares the predicted probability of each class with actual class output. These values are 0 and 1. This scoring system applies penalties to predicted probability depending on the distance to the expected value. Since the function uses logarithms, small distance has less impact and large distance has more impact.

The cross-entropy loss is used to approximate the true distribution of target variable, since the applied penalty is different with small differences and large differences. By minimizing the cross-entropy loss, the model learns to assign higher probabilities to correct outputs, which improves the classification performance.

## **Gradient Computation**

From definition of SGD approach:

Therefore

## **Some Training Parameters and Basic Parameter Calculations**

### **What are batch and epoch in the context of MLP training?**

Batch: Batch is a subset of the training data which is used to compute the gradient of the loss function with respect to the model parameters.

Epoch: Epoch is a complete pass through the entire training dataset in the algorithm. During each epoch, the model is trained on all the training examples in the dataset.

### **Given that the dataset has N samples, what is the number of batches per epoch if the batch size is B?**

# of batches per epoch = ceil(N/B), where ceil is the ceiling function to round the result of N/B to the nearest integer.

### **Given that the dataset has N samples, what is the number of SGD iterations if you want to train your ANN for E epochs with the batch size of B?**

# of SGD iterations = E \* (# of batches per epoch) = E \* (ceil(N/B))

## **Computing Number of Parameters of ANN Classifiers**

### **Consider an MLP classifier of K hidden units where the size of each hidden unit is Hk for k=1, . . ., K. Derive a formula to compute the number of parameters that the MLP has if the input and output dimensions are Din and Dout, respectively.**

Input Layer: Input layer has Din units and each unit is connected to the first hidden layer H1, which results with Din\*H1 connections. Also, there is a bias term H1. In conclusion, parameters formula from first layer is Din\*H1+H1

Hidden Layers: For each hidden layer k=2,…,K, the previous layer has Hk-1 units and current layer has Hk units. When these connections are made with including the bias Hk, the result from hidden layers is   
Hk-1\*Hk + Hk

Output Layer: Output layer has Dout units, and last hidden layer Hk is connected to output layer. When these connections are made with including the bias Dout, the result from output layer is Dout\*Hk+Dout

When all of these parameters are combined, the result is:

### **Consider a CNN classifier of K convolutional layers where the spatial size of each layer is Hk×Wk and the number of convolutional filters (kernels) of each layer is Ck for k=1, . . ., K. Derive a formula to compute the number of parameters that the CNN has if the input dimension is Hin×Win****×Cin.**

For each convolutional layer k=1,…,K, the number of parameters can be found by Hk×Wk×Ck×Ck-1+Ck. For k=1, instead of Ck-1=0, we can use Cin. Hin and Win is not in the number of parameters formula since they don’t have an impact on learnable parameters.

# Implementing a Convolutional Layer with NumPy

## Experimental Work (Code in Appendix A)

A picture containing text, fabric

Description automatically generated

Figure Result of my\_conv2d.py

## 2.2. Discussions

### 2.2.1. Why are Convolutional Neural Networks important? Why are they used in image processing?

Convolutional Neural Networks are important for image processing because they can automate the feature extraction process and can learn to recognize complex patterns and structures in images. This makes them highly useful for tasks such as image classification, object detection, and image segmentation.

### 2.2.2. What is a kernel of a Convolutional Layer? What do the sizes of a kernel correspond to?

In Convolutional Neural Networks, the kernel of a convolutional layer is a small matrix of weights that slides over the input image. The kernel is also known as a filter, and it's responsible for performing the dot product operation at each location.

The size of the kernel corresponds to the area of the input image that a single neuron in the convolutional layer is sensitive to. For example, if the kernel size is 5x5, each neuron in the convolutional layer is sensitive to a 5x5 patch of pixels in the input image.

### 2.2.3. Briefly explain the output image. What happened here?

The output image is the 2D convolution operation on the input using a kernel, which basically is a filter with small matrix of weights. Input image is converted to black and white pixels with transformed feature space.

### 2.2.4. Why are the numbers in the same column look alike to each other, even though they belong to different images?

The numbers in the same column look alike to each other because they represent the output values of the same kernel that was applied to different patches. When the kernel is applied to different patches, it produces similar output values because those patches may contain similar features that kernel was designed to detect. Since the output values are not normalized in this code, they are like each other more. Applying ReLU or different normalization techniques can solve this issue.

### 2.2.5. Why are the numbers in the same row do not look alike to each other, even though they belong to same image?

The numbers in the same row do not look alike to each other because they represent the output values of different patches. The input image is divided into multiple small patches, and each patch is processed independently by the same set kernels. The patches may capture different features or patterns of the image and they produce different output features in each patch. Also, the order of the patches in the output image may affect the appearance of the output values.

### 2.2.6. What can be deduced about Convolutional Layers from your answers to Questions 4 and 5?

It can be deduced that, convolutional layers use kernels to detect specific features in from input, and when a kernel is applied to different patches of the input, it produces similar output values for patches that contain similar features. Also, when different kernels are applied to the same patch of the input, they produce different output values that reflect the features in that patch.

# Experimenting ANN Architectures

## Experimental Work (Code in Appendix B)

Graphical user interface

Description automatically generated

Figure Result Plots of Experimental Work 3.1.

Table

Description automatically generated with low confidence

Figure input\_weights\_mlp\_1

Table

Description automatically generated

Figure input\_weights\_mlp\_2

A picture containing crossword puzzle, indoor, text, black

Description automatically generated

Figure input\_weights\_cnn\_3

A picture containing crossword puzzle, indoor, text, tiled

Description automatically generated

Figure input\_weights\_cnn\_4

A picture containing crossword puzzle

Description automatically generated

Figure input\_weights\_cnn\_5

## Discussions

### What is the generalization performance of a classifier?

The generalization performance of a classifier is the ability of a model to make good predictions on data that is not trained before. For this homework, we check generalization performance with validation and test data, which is not trained since they are separated from train data. If the results are satisfying, the training is sufficient for new data classification. If the results are not satisfying, there can be overfit with training data and new data may not be generalized well.

### Which plots are informative to inspect generalization performance?

To find information on generalization performance, we should be able to see how the trained model performs on data which it didn’t see before. The validation accuracy plot shows how well the model is performing on new data and test vs training accuracy plot provides an indication of how well the model is able to generalize to new data.

### Compare the generalization performance of the architectures.

mlp\_1 and mlp\_2 have the worst generalization performance when the test vs train accuracy plot is checked, meanwhile cnn\_3, cnn\_4 and cnn\_5 have better results where cnn\_4 has the best generalization performance.

### How does the number of parameters affect the classification and generalization performance?

Increasing the number of parameters can make the model better fit the training data, creating a higher classification performance on the training set. However, too many number of parameters can cause the model to overfit to the training data, which results with lower generalization performance on the validation and test sets.

### How does the depth of the architecture affect the classification and generalization performance?

Increasing the depth of a deep neural network architecture can allow it to learn more complex data, creating a higher classification performance on the training set. However, if the network is too deep, it can be difficult to train because gradients can disappear, or overfitting can happen. This can result in lower generalization and classification performance on the validation and test sets.

### Considering the visualizations of the weights, are they interpretable?

The visualizations of the weights are somehow interpretable when linear regression is applied, where each weight represents the impact of a specific input feature. For our case, since we create the visualization of the first layer weights only, the weights can be difficult to interpret directly.

### Can you say whether the units are specialized to specific classes?

Since we only have the weight visualizations from first layer and do not include general structure of the model, it is not possible to decide if the units are specialized to specific classes.

### Weights of which architecture are more interpretable?

In our trainings, MLP models provide more information in weight visualizations since they have more details in them.

### Considering the architectures, comment on the structures (how they are designed). Can you say that some architecture are akin to each other? Compare the performance of similarly structured architectures and architectures with different structure

mlp\_1 and mlp\_2 are similar to each other, which both are simple feedforward neural networks. mlp\_2 has more hidden layers compared to mlp\_1, which provides improved performance. This architecture can be used for simple classification tasks. As the complexity increases, the performance drops.

cnn\_4 is extension of cnn\_3 with additional convolutional layers and pooling layers, and provides better performance. Also, cnn\_5 is extension of cnn\_4 with additional connections between convolutional layers. All of them are convolutional neural networks which handle image classifications and input structure has a spatial structure. cnn\_4 provides better accuracy although cnn\_5 has more complexity. This can be due to the training time or parameter tuning.

When MLP and CNN models are compared, CNN provides better accuracy, and this is due to the complexity of the dataset we have.

### Which architecture would you pick for this classification task? Why?

I would choose cnn\_4 as it provides better results, and it might have better parameter settings.

# Experimenting Activation Functions

## Experimental Work (Code in Appendix C)

Chart, histogram

Description automatically generated Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated

Figure Result Plots from Experimental Work 4.1.

## Discussions

### How is the gradient behavior in different architectures? What happens when depth increases?

For all of the architectures, training losses are higher with sigmoid function and gradient losses are lower with sigmoid function. ReLU has constant gradients for all inputs, which helps to have gradients in deeper levels also and they do not disappear. However, sigmoid function assigns a logistic gradient and as the model gets deeper, the gradients disappear.

### Why do you think that happens?

It happens due to the depth makes the model more complex and weight gradient have more relations between each other. ReLU sets negative values to zero and effectively ignore their effect, which makes deeper layers more feasible. On the other hand, sigmoid function bounds the outputs between 0 to 1 and this causes the gradient to be very small or vanish. This makes training deeper networks harders since the gradient updates become smaller.

### What might happen if we use inputs in the range [0, 255] instead of [0.0, 1.0]?

If we use inputs in the range [0, 255] instead of [0.0, 1.0], the training will have slower convergence due to the weights in neural networks, which are initialized randomly and the initial range of weights is generally small. When the inputs are in the range of [0, 255], the dot product of the weights and inputs can be much larger. This can cause the gradients becoming very small, which can make it harder for the model to learn. To not face that, we normalize the input to [0.0, 1.0], to have dot product of weights and inputs within a reasonable range.

# Experimenting Learning Rate

## Experimental Work (Code in Appendix D,E,F and G)

Chart, line chart, histogram

Description automatically generatedChart, histogram

Description automatically generatedChart, histogram

Description automatically generated

Figure Result Plots from Experimental Work 5.1.

## Discussions

### How does the learning rate affect the convergence speed?

As the learning rate increases, there will be a faster convergence. However, the risk of overshooting the minimum of the loss function also increases.

### How does the learning rate affect the convergence to a better point?

The learning rate controls the step size during the optimization process and affects the convergence speed. A higher learning rate requires less epochs, but this can result in worse training results. A lower learning rate requires more epochs, and this can result with stuck in a state because the optimizer may get trapped in local minima. This can result in searching for global minimum.

### Does your scheduled learning rate method work? In what sense?

It worked because when it is compared with the result that we did not touch, the validation accuracy had more stable results with our adjustment.

### Compare the accuracy and convergence performance of your scheduled learning rate method with Adam.

With scheduled learning rate with Adam, I achieved a test accuracy of 59% and with Adam optimizer only, I had 47% test accuracy. With help of scheduled learning rate, we had better convergence performance.

# Appendix

## Code Snippet of 2.1.

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

from torchvision.utils import make\_grid

def part2Plots(out, nmax=64, save\_dir='', filename=''):

    out = torch.tensor(out).reshape(-1,1,25,25)

    fig, ax = plt.subplots(figsize=(8, 8))

    ax.set\_xticks([]); ax.set\_yticks([])

    ax.imshow(make\_grid((out.detach()[:nmax]), nrow=8).permute(1, 2, 0))

    fig.savefig(os.path.join(save\_dir, filename + '.png'))

def my\_conv2d(input, kernel):

*# Get input and kernel shapes*

    batch\_size, input\_channels, input\_height, input\_width = input.shape

    output\_channels, input\_channels, filter\_height, filter\_width = kernel.shape

*# Compute output tensor shape*

    output\_height = input\_height - filter\_height + 1

    output\_width = input\_width - filter\_width + 1

*# Initialize output tensor with zeros*

    output = np.zeros((batch\_size, output\_channels, output\_height, output\_width))

*# Loop over batch, output channels, and output spatial dimensions*

    for b in range(batch\_size):

        for c\_out in range(output\_channels):

            for i in range(output\_height):

                for j in range(output\_width):

*# Compute dot product between input patch and kernel patch*

                    output[b, c\_out, i, j] = np.sum(

                        input[b, :, i:i+filter\_height, j:j+filter\_width] \* kernel[c\_out]

                    )

    return output

def hw1\_2():

*#input shape: [batch\_size, input\_Channels, input\_height, input\_width]*

    input = np.load('samples\_2.npy')

*#input shape: [output\_channels, input\_Channels, filter\_height, filter\_width]*

    kernel = np.load('kernel.npy')

    out = my\_conv2d(input, kernel)

    part2Plots(out, save\_dir = '.', filename = 'output')

hw1\_2()

## Code Snippet of 3.1.

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

import torchvision

import json

from utils import visualizeWeights, part3Plots

import json

from sklearn.model\_selection import train\_test\_split

*# Parameter Setting*

BATCH\_SIZE = 50

NUM\_EPOCHS = 15

NUM\_STEPS = 10

LEARNING\_RATE = 0.01

NUM\_CLASSES = 10

*# Class Definitions*

class mlp\_1(torch.nn.Module):

    def \_\_init\_\_(self, input\_size, num\_classes):

        super(mlp\_1, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.fc1 = torch.nn.Linear(input\_size, 32)

        self.relu = torch.nn.ReLU()

        self.fc2 = torch.nn.Linear(32, num\_classes)

    def forward(self, x):

        x = x.view(-1, self.input\_size)

        x = self.fc1(x)

        x = self.relu(x)

        x = self.fc2(x)

        return x

class mlp\_2(torch.nn.Module):

    def \_\_init\_\_(self, input\_size, num\_classes):

        super(mlp\_2, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.fc1 = torch.nn.Linear(input\_size, 32)

        self.relu = torch.nn.ReLU()

        self.fc2 = torch.nn.Linear(32, 64, bias=False)

        self.fc3 = torch.nn.Linear(64, num\_classes)

    def forward(self, x):

        x = x.view(-1, self.input\_size)

        x = self.fc1(x)

        x = self.relu(x)

        x = self.fc2(x)

        x = self.fc3(x)

        return x

class cnn\_3(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

class cnn\_4(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_4, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=5, stride=1, padding='valid')

        self.relu3 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv4 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=16, kernel\_size=5, stride=1, padding='valid')

        self.relu4 = torch.nn.ReLU()

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 4 \* 4, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.conv3(x)

        x = self.relu3(x)

        x = self.maxpool1(x)

        x = self.conv4(x)

        x = self.relu4(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 4 \* 4)

        x = self.fc(x)

        return x

class cnn\_5(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_5, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.conv3 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, padding='valid')

        self.relu3 = torch.nn.ReLU()

        self.conv4 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu4 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv5 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu5 = torch.nn.ReLU()

        self.conv6 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.relu6 = torch.nn.ReLU()

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(8 \* 4 \* 4, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.conv3(x)

        x = self.relu3(x)

        x = self.conv4(x)

        x = self.relu4(x)

        x = self.maxpool1(x)

        x = self.conv5(x)

        x = self.relu5(x)

        x = self.conv6(x)

        x = self.relu6(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 8 \* 4 \* 4)

        x = self.fc(x)

        return x

*# device selection*

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"{device} is available")

print(torch.cuda.is\_available())

print(torch.cuda.current\_device())

print(torch.cuda.device(0))

print(torch.cuda.device\_count())

print(torch.cuda.get\_device\_name(0))

*# hyperparameters*

transform = torchvision.transforms.Compose([

    torchvision.transforms.ToTensor(),

    torchvision.transforms.Normalize(

        (0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),

    torchvision.transforms.Grayscale()

])

*# training set*

trainset = torchvision.datasets.CIFAR10(

    './data', train=True, download=True, transform=transform)

*# train set and validation set*

trainset, valset = train\_test\_split(trainset, test\_size=0.1, random\_state=42)

testset = torchvision.datasets.CIFAR10(

    './data', train=False, transform=transform)

*# data loader for training set, validation set and test set*

trainloader = torch.utils.data.DataLoader(

    trainset, batch\_size=BATCH\_SIZE, shuffle=True)

valloader = torch.utils.data.DataLoader(

    valset, batch\_size=BATCH\_SIZE, shuffle=False)

testloader = torch.utils.data.DataLoader(

    testset, batch\_size=BATCH\_SIZE, shuffle=False)

classes = ('airplane', 'automobile', 'bird', 'cat',

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

plots = []

*# model loop*

for ctr in range(5):

    model\_train\_loss = []

    model\_train\_acc = []

    model\_val\_acc = []

    best\_weight = 0

    best\_acc = 0

*# step loop*

    for step in range(NUM\_STEPS):

        if ctr == 0:

            model = mlp\_1(1024, 10)

        elif ctr == 1:

            model = mlp\_2(1024, 10)

        elif ctr == 2:

            model = cnn\_3(10)

        elif ctr == 3:

            model = cnn\_4(10)

        elif ctr == 4:

            model = cnn\_5(10)

        model = model.to(device)

        model\_name = model.\_\_class\_\_.\_\_name\_\_

*# loss function and optimizer*

        optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING\_RATE)

        criterion = torch.nn.CrossEntropyLoss().to(device)

*# step lists*

        step\_train\_loss = []

        step\_train\_acc = []

        step\_val\_acc = []

        step\_test\_acc = []

*# epoch loop*

        for epoch in range(NUM\_EPOCHS):

            print(

                f'Epoch {epoch+1}/{NUM\_EPOCHS}  Step {step+1}/{NUM\_STEPS} for {model\_name} model')

*# training loader*

            trainloader = torch.utils.data.DataLoader(

                trainset, batch\_size=BATCH\_SIZE, shuffle=True)

*# training loop*

            for i, tr\_data in enumerate(trainloader, 0):

                model.train()

                tr\_inp, tr\_lab = tr\_data[0].to(device), tr\_data[1].to(device)

*# forward + backward + optimize*

                tr\_out = model(tr\_inp)

                tr\_loss = criterion(tr\_out, tr\_lab)

                optimizer.zero\_grad()

                tr\_loss.backward()

                optimizer.step()

*# print statistics each 10 steps*

                if i % 10 == 9:

                    model.eval()

                    \_, tr\_pred = tr\_out.max(1)

                    n\_samples = tr\_lab.size(0)

                    n\_correct = (tr\_pred == tr\_lab).sum().item()

*# training accuracy*

                    training\_acc = 100.0 \* n\_correct / n\_samples

                    train\_loss = tr\_loss.item()

                    val\_total = 0

                    val\_correct = 0

*# validation loop*

                    for j, val\_data in enumerate(valloader, 0):

                        val\_inp, val\_lab = val\_data[0].to(

                            device), val\_data[1].to(device)

                        val\_out = model(val\_inp)

                        \_, val\_pred = val\_out.max(1)

                        val\_total += val\_lab.size(0)

                        val\_correct += (val\_pred ==

                                        val\_lab).sum().item()

*# validation accuracy*

                    val\_acc = 100.0 \* val\_correct / val\_total

*# record statistics*

                    step\_train\_loss.append(train\_loss)

                    step\_train\_acc.append(training\_acc)

                    step\_val\_acc.append(val\_acc)

*# record statistics*

        model\_train\_loss.append(step\_train\_loss)

        model\_train\_acc.append(step\_train\_acc)

        model\_val\_acc.append(step\_val\_acc)

*# test loop*

        with torch.no\_grad():

            model.eval()

            testloader = torch.utils.data.DataLoader(

                testset, batch\_size=BATCH\_SIZE, shuffle=False)

            n\_correct = 0

            n\_samples = 0

            for test\_images, test\_labels in testloader:

                test\_images, test\_labels = test\_images.to(

                    device), test\_labels.to(device)

                test\_outputs = model(test\_images)

                \_, test\_predicted = test\_outputs.max(1)

                n\_samples += test\_labels.size(0)

                n\_correct += (test\_predicted == test\_labels).sum().item()

*# test accuracy*

            test\_acc = 100.0 \* n\_correct / n\_samples

            step\_test\_acc.append(test\_acc)

*# save best model*

            if (test\_acc > best\_acc):

                best\_acc = test\_acc

                model.to('cpu')

                if (model.\_\_class\_\_.\_\_name\_\_ == 'mlp\_1' or model.\_\_class\_\_.\_\_name\_\_ == 'mlp\_2'):

                    best\_weight = model.fc1.weight.data.numpy()

                else:

                    best\_weight = model.conv1.weight.data.numpy()

                model.to(device)

*# average statistics*

    avg\_train\_loss = [sum(x)/len(x) for x in zip(\*model\_train\_loss)]

    avg\_train\_acc = [sum(x)/len(x) for x in zip(\*model\_train\_acc)]

    avg\_valid\_acc = [sum(x)/len(x) for x in zip(\*model\_val\_acc)]

    model\_result = {

        'name': model\_name,

        'loss\_curve': avg\_train\_loss,

        'train\_acc\_curve': avg\_train\_acc,

        'val\_acc\_curve': avg\_valid\_acc,

        'test\_acc': best\_acc,

        'weights': best\_weight.tolist(),

    }

*# save model results*

    with open("Q3\_JSON/Q3\_"+model\_name+".json", "w") as outfile:

        json.dump(model\_result, outfile)

*# save model weights*

    visualizeWeights(best\_weight, save\_dir='Q3\_Images',

                     filename='input\_weights\_'+model\_name)

*# save plots*

    plots.append(json.load(morel\_result))

    part3Plots(plots, save\_dir=r'Q3\_Images', filename='part3Plots')

print("Training Done")

## Code Snippet of 4.1.

from utils import part4Plots

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

import torchvision

import json

from utils import visualizeWeights, part4Plots

from sklearn.model\_selection import train\_test\_split

*# parameters*

BATCH\_SIZE = 50

NUM\_EPOCHS = 15

NUM\_STEPS = 1

LEARNING\_RATE = 0.01

NUM\_CLASSES = 10

*# class definitions*

class mlp\_1\_relu(torch.nn.Module):

    def \_\_init\_\_(self, input\_size, num\_classes):

        super(mlp\_1\_relu, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.fc1 = torch.nn.Linear(input\_size, 32)

        self.relu = torch.nn.ReLU()

        self.fc2 = torch.nn.Linear(32, num\_classes)

    def forward(self, x):

        x = x.view(-1, self.input\_size)

        x = self.fc1(x)

        x = self.relu(x)

        x = self.fc2(x)

        return x

class mlp\_1\_sigmo(torch.nn.Module):

    def \_\_init\_\_(self, input\_size, num\_classes):

        super(mlp\_1\_sigmo, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.fc1 = torch.nn.Linear(input\_size, 32)

        self.sigmo = torch.nn.Sigmoid()

        self.fc2 = torch.nn.Linear(32, num\_classes)

    def forward(self, x):

        x = x.view(-1, self.input\_size)

        x = self.fc1(x)

        x = self.sigmo(x)

        x = self.fc2(x)

        return x

class mlp\_2\_relu(torch.nn.Module):

    def \_\_init\_\_(self, input\_size, num\_classes):

        super(mlp\_2\_relu, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.fc1 = torch.nn.Linear(input\_size, 32)

        self.relu = torch.nn.ReLU()

        self.fc2 = torch.nn.Linear(32, 64, bias=False)

        self.fc3 = torch.nn.Linear(64, num\_classes)

    def forward(self, x):

        x = x.view(-1, self.input\_size)

        x = self.fc1(x)

        x = self.relu(x)

        x = self.fc2(x)

        x = self.fc3(x)

        return x

class mlp\_2\_sigmo(torch.nn.Module):

    def \_\_init\_\_(self, input\_size, num\_classes):

        super(mlp\_2\_sigmo, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.fc1 = torch.nn.Linear(input\_size, 32)

        self.sigmo = torch.nn.Sigmoid()

        self.fc2 = torch.nn.Linear(32, 64, bias=False)

        self.fc3 = torch.nn.Linear(64, num\_classes)

    def forward(self, x):

        x = x.view(-1, self.input\_size)

        x = self.fc1(x)

        x = self.sigmo(x)

        x = self.fc2(x)

        x = self.fc3(x)

        return x

class cnn\_3\_relu(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3\_relu, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

class cnn\_3\_sigmo(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3\_sigmo, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.sigmo1 = torch.nn.Sigmoid()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.sigmo2 = torch.nn.Sigmoid()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.sigmo1(x)

        x = self.conv2(x)

        x = self.sigmo2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

class cnn\_4\_relu(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_4\_relu, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=5, stride=1, padding='valid')

        self.relu3 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv4 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=16, kernel\_size=5, stride=1, padding='valid')

        self.relu4 = torch.nn.ReLU()

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 4 \* 4, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.conv3(x)

        x = self.relu3(x)

        x = self.maxpool1(x)

        x = self.conv4(x)

        x = self.relu4(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 4 \* 4)

        x = self.fc(x)

        return x

class cnn\_4\_sigmo(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_4\_sigmo, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.sigmo1 = torch.nn.Sigmoid()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.sigmo2 = torch.nn.Sigmoid()

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=5, stride=1, padding='valid')

        self.sigmo3 = torch.nn.Sigmoid()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv4 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=16, kernel\_size=5, stride=1, padding='valid')

        self.sigmo4 = torch.nn.Sigmoid()

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 4 \* 4, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.sigmo1(x)

        x = self.conv2(x)

        x = self.sigmo2(x)

        x = self.conv3(x)

        x = self.sigmo3(x)

        x = self.maxpool1(x)

        x = self.conv4(x)

        x = self.sigmo4(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 4 \* 4)

        x = self.fc(x)

        return x

class cnn\_5\_relu(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_5\_relu, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.conv3 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, padding='valid')

        self.relu3 = torch.nn.ReLU()

        self.conv4 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu4 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv5 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu5 = torch.nn.ReLU()

        self.conv6 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.relu6 = torch.nn.ReLU()

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(8 \* 4 \* 4, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.conv3(x)

        x = self.relu3(x)

        x = self.conv4(x)

        x = self.relu4(x)

        x = self.maxpool1(x)

        x = self.conv5(x)

        x = self.relu5(x)

        x = self.conv6(x)

        x = self.relu6(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 8 \* 4 \* 4)

        x = self.fc(x)

        return x

class cnn\_5\_sigmo(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_5\_sigmo, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.sigmo1 = torch.nn.Sigmoid()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.sigmo2 = torch.nn.Sigmoid()

        self.conv3 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, padding='valid')

        self.sigmo3 = torch.nn.Sigmoid()

        self.conv4 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.sigmo4 = torch.nn.Sigmoid()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv5 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.sigmo5 = torch.nn.Sigmoid()

        self.conv6 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=3, stride=1, padding='valid')

        self.sigmo6 = torch.nn.Sigmoid()

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(8 \* 4 \* 4, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.sigmo1(x)

        x = self.conv2(x)

        x = self.sigmo2(x)

        x = self.conv3(x)

        x = self.sigmo3(x)

        x = self.conv4(x)

        x = self.sigmo4(x)

        x = self.maxpool1(x)

        x = self.conv5(x)

        x = self.sigmo5(x)

        x = self.conv6(x)

        x = self.sigmo6(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 8 \* 4 \* 4)

        x = self.fc(x)

        return x

*# device setting*

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"{device} is available")

print(torch.cuda.is\_available())

print(torch.cuda.current\_device())

print(torch.cuda.device(0))

print(torch.cuda.device\_count())

print(torch.cuda.get\_device\_name(0))

*# transform*

transform = torchvision.transforms.Compose([

    torchvision.transforms.ToTensor(),

    torchvision.transforms.Normalize(

        (0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),

    torchvision.transforms.Grayscale()

])

*# training set*

trainset = torchvision.datasets.CIFAR10(

    './data', train=True, download=True, transform=transform)

*# train-val split*

trainset, valset = train\_test\_split(trainset, test\_size=0.1, random\_state=42)

testset = torchvision.datasets.CIFAR10(

    './data', train=False, transform=transform)

*# dataloader for training, test, validation*

trainloader = torch.utils.data.DataLoader(

    trainset, batch\_size=BATCH\_SIZE, shuffle=True)

valloader = torch.utils.data.DataLoader(

    valset, batch\_size=BATCH\_SIZE, shuffle=False)

testloader = torch.utils.data.DataLoader(

    testset, batch\_size=BATCH\_SIZE, shuffle=False)

classes = ('airplane', 'automobile', 'bird', 'cat',

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

*# data list*

relu\_loss\_curve = []

sigmoid\_loss\_curve = []

relu\_grad\_curve = []

sigmoid\_grad\_curve = []

plots = []

*# model looper*

for ctr in range(10):

    for step in range(NUM\_STEPS):

        if ctr == 0:

            model = mlp\_1\_relu(1024, 10)

        elif ctr == 1:

            model = mlp\_1\_sigmo(1024, 10)

        elif ctr == 2:

            model = mlp\_2\_relu(1024, 10)

        elif ctr == 3:

            model = mlp\_2\_sigmo(1024, 10)

        elif ctr == 4:

            model = cnn\_3\_relu(10)

        elif ctr == 5:

            model = cnn\_3\_sigmo(10)

        elif ctr == 6:

            model = cnn\_4\_relu(10)

        elif ctr == 7:

            model = cnn\_4\_sigmo(10)

        elif ctr == 8:

            model = cnn\_5\_relu(10)

        elif ctr == 9:

            model = cnn\_5\_sigmo(10)

        model = model.to(device)

        model\_name = model.\_\_class\_\_.\_\_name\_\_

*# optimizer and loss function*

        optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING\_RATE)

        criterion = torch.nn.CrossEntropyLoss().to(device)

*# epoch loop*

        for epoch in range(NUM\_EPOCHS):

            print(

                f'Epoch {epoch+1}/{NUM\_EPOCHS}  Step {step+1}/{NUM\_STEPS} for {model\_name} model')

*# train loader*

            trainloader = torch.utils.data.DataLoader(

                trainset, batch\_size=BATCH\_SIZE, shuffle=True)

*# train loop*

            for i, tr\_data in enumerate(trainloader, 0):

                model.train()

                tr\_inp, tr\_lab = tr\_data[0].to(device), tr\_data[1].to(device)

                model.to('cpu')

*# get gradient for mlp and cnn*

                if (ctr <= 3):

                    gradA = model.fc1.weight.data.numpy().flatten()

                else:

                    gradA = model.conv1.weight.data.numpy().flatten()

                model.to(device)

*# forward + backward + optimize*

                tr\_out = model(tr\_inp)

                tr\_loss = criterion(tr\_out, tr\_lab)

                optimizer.zero\_grad()

                tr\_loss.backward()

                optimizer.step()

*# record statistics every 10 steps for gradient and loss*

                if i % 10 == 9:

                    model.to('cpu')

*# record conditions for mlp and cnn*

                    if (ctr <= 3):

                        gradB = model.fc1.weight.data.numpy().flatten()

                        grad\_magnitude = float(np.linalg.norm(gradA - gradB))

                    else:

                        gradB = model.conv1.weight.data.numpy().flatten()

                        grad\_magnitude = float(np.linalg.norm(gradA - gradB))

                    if (ctr == 0 or ctr == 2 or ctr == 4 or ctr == 6 or ctr == 8):

                        relu\_grad\_curve.append(grad\_magnitude)

                        relu\_loss\_curve.append(tr\_loss.item())

                    else:

                        sigmoid\_grad\_curve.append(grad\_magnitude)

                        sigmoid\_loss\_curve.append(tr\_loss.item())

                    model.to(device)

*# only record every 2nd loop since we have 10 models of doubles*

    if (ctr % 2 == 1):

        model\_result = {

            'name': model\_name[0:5],

            'relu\_loss\_curve': relu\_loss\_curve,

            'sigmoid\_loss\_curve': sigmoid\_loss\_curve,

            'relu\_grad\_curve': relu\_grad\_curve,

            'sigmoid\_grad\_curve': sigmoid\_grad\_curve,

        }

*# clean up for next double model*

        relu\_loss\_curve = []

        sigmoid\_loss\_curve = []

        relu\_grad\_curve = []

        sigmoid\_grad\_curve = []

*# save json*

        with open("Q4\_JSON/Q4\_"+model\_name[0:5]+".json", "w") as outfile:

            json.dump(model\_result, outfile)

*# plot*

    plots.append(json.load(model\_result))

    part4Plots(plots, save\_dir=r'Q4\_IMAGES', filename=model+"\_plot")

print("Training Done")

## Code Snippet of 5.1.

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

import torchvision

import json

from utils import visualizeWeights, part5Plots

from sklearn.model\_selection import train\_test\_split

*# parameters*

BATCH\_SIZE = 50

NUM\_EPOCHS = 20

NUM\_STEPS = 1

LEARNING\_RATE = [0.1, 0.01, 0.001]

NUM\_CLASSES = 10

*# class definition*

class cnn\_3(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

*# device setup*

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"{device} is available")

print(torch.cuda.is\_available())

print(torch.cuda.current\_device())

print(torch.cuda.device(0))

print(torch.cuda.device\_count())

print(torch.cuda.get\_device\_name(0))

*# transform*

transform = torchvision.transforms.Compose([

    torchvision.transforms.ToTensor(),

    torchvision.transforms.Normalize(

        (0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),

    torchvision.transforms.Grayscale()

])

*# training set*

trainset = torchvision.datasets.CIFAR10(

    './data', train=True, download=True, transform=transform)

*# train set split*

trainset, valset = train\_test\_split(trainset, test\_size=0.1, random\_state=42)

testset = torchvision.datasets.CIFAR10(

    './data', train=False, transform=transform)

*# loader for training, validation and test set*

trainloader = torch.utils.data.DataLoader(

    trainset, batch\_size=BATCH\_SIZE, shuffle=True)

valloader = torch.utils.data.DataLoader(

    valset, batch\_size=BATCH\_SIZE, shuffle=False)

testloader = torch.utils.data.DataLoader(

    testset, batch\_size=BATCH\_SIZE, shuffle=False)

classes = ('airplane', 'automobile', 'bird', 'cat',

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

*# list initialization*

tra\_loss\_curve = []

val\_acc\_curve = []

*# loop for different learning rates*

for ctr in LEARNING\_RATE:

    train\_loss\_lr = []

    val\_acc\_lr = []

    model = cnn\_3(10)

    model = model.to(device)

    model\_name = model.\_\_class\_\_.\_\_name\_\_

*# optimizer and loss function*

    optimizer = torch.optim.SGD(model.parameters(), lr=ctr)

    criterion = torch.nn.CrossEntropyLoss().to(device)

*# epoch loop*

    for epoch in range(NUM\_EPOCHS):

        print(

            f'Epoch {epoch+1}/{NUM\_EPOCHS}  lr = {ctr} for {model\_name} model')

*# train loader*

        trainloader = torch.utils.data.DataLoader(

            trainset, batch\_size=BATCH\_SIZE, shuffle=True)

*# batch loop*

        for i, tr\_data in enumerate(trainloader, 0):

            model.train()

            tr\_inp, tr\_lab = tr\_data[0].to(device), tr\_data[1].to(device)

*# forward + backward + optimize*

            tr\_out = model(tr\_inp)

            tr\_loss = criterion(tr\_out, tr\_lab)

            optimizer.zero\_grad()

            tr\_loss.backward()

            optimizer.step()

*# record every 10th step*

            if i % 10 == 9:

                model.eval()

                \_, tr\_pred = tr\_out.max(1)

                n\_samples = tr\_lab.size(0)

                n\_correct = (tr\_pred == tr\_lab).sum().item()

                training\_acc = 100.0 \* n\_correct / n\_samples

                train\_loss = tr\_loss.item()

                val\_total = 0

                val\_correct = 0

*# validation loop*

                for j, val\_data in enumerate(valloader, 0):

                    val\_inp, val\_lab = val\_data[0].to(

                        device), val\_data[1].to(device)

                    val\_out = model(val\_inp)

                    \_, val\_pred = val\_out.max(1)

                    val\_total += val\_lab.size(0)

                    val\_correct += (val\_pred ==

                                    val\_lab).sum().item()

                val\_acc = 100.0 \* val\_correct / val\_total

                train\_loss\_lr.append(train\_loss)

                val\_acc\_lr.append(val\_acc)

    tra\_loss\_curve.append(train\_loss\_lr)

    val\_acc\_curve.append(val\_acc\_lr)

*# save the results*

model\_result = {

    'name': model\_name,

    'loss\_curve\_1': tra\_loss\_curve[0],

    'loss\_curve\_01': tra\_loss\_curve[1],

    'loss\_curve\_001': tra\_loss\_curve[2],

    'val\_acc\_curve\_1': val\_acc\_curve[0],

    'val\_acc\_curve\_01': val\_acc\_curve[1],

    'val\_acc\_curve\_001': val\_acc\_curve[2]

}

*# save the results as json file*

with open("Q5\_JSON/Q5\_"+"learning"+".json", "w") as outfile:

    json.dump(model\_result, outfile)

*# plot the results*

part5Plots(model\_result, save\_dir=r'Q5\_IMAGES', filename="learning")

print("Training Done")

## Code Snippet of 5.1.5.

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

import torchvision

import json

from utils import visualizeWeights

from sklearn.model\_selection import train\_test\_split

*# parameters*

BATCH\_SIZE = 50

NUM\_EPOCHS = 30

NUM\_STEPS = 1

LEARNING\_RATE = 0.1

NUM\_CLASSES = 10

LIMIT = 1000

*# class definition*

class cnn\_3(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

*# device selection*

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"{device} is available")

print(torch.cuda.is\_available())

print(torch.cuda.current\_device())

print(torch.cuda.device(0))

print(torch.cuda.device\_count())

print(torch.cuda.get\_device\_name(0))

*# transform definition*

transform = torchvision.transforms.Compose([

    torchvision.transforms.ToTensor(),

    torchvision.transforms.Normalize(

        (0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),

    torchvision.transforms.Grayscale()

])

*# training set*

trainset = torchvision.datasets.CIFAR10(

    './data', train=True, download=True, transform=transform)

*# trainset split*

trainset, valset = train\_test\_split(trainset, test\_size=0.1, random\_state=42)

testset = torchvision.datasets.CIFAR10(

    './data', train=False, transform=transform)

*# data loader for training, validation and test sets*

trainloader = torch.utils.data.DataLoader(

    trainset, batch\_size=BATCH\_SIZE, shuffle=True)

valloader = torch.utils.data.DataLoader(

    valset, batch\_size=BATCH\_SIZE, shuffle=False)

testloader = torch.utils.data.DataLoader(

    testset, batch\_size=BATCH\_SIZE, shuffle=False)

classes = ('airplane', 'automobile', 'bird', 'cat',

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

*# curve list initialization*

tra\_loss\_curve = []

val\_acc\_curve = []

*# loss and acc lr list initialization*

train\_loss\_lr = []

val\_acc\_lr = []

model = cnn\_3(10)

model = model.to(device)

model\_name = model.\_\_class\_\_.\_\_name\_\_

*# optimizer and loss function definition*

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING\_RATE)

criterion = torch.nn.CrossEntropyLoss().to(device)

*# ctr for lr update*

ctr = 0

*# epoch loop*

for epoch in range(NUM\_EPOCHS):

    print(

        f'Epoch {epoch+1}/{NUM\_EPOCHS}  lr = {LEARNING\_RATE} for {model\_name} model')

    trainloader = torch.utils.data.DataLoader(

        trainset, batch\_size=BATCH\_SIZE, shuffle=True)

*# batch loop*

    for i, tr\_data in enumerate(trainloader, 0):

        model.train()

        tr\_inp, tr\_lab = tr\_data[0].to(device), tr\_data[1].to(device)

*# forward + backward + optimize*

        tr\_out = model(tr\_inp)

        tr\_loss = criterion(tr\_out, tr\_lab)

        optimizer.zero\_grad()

        tr\_loss.backward()

        optimizer.step()

*# record statistich each 10 batch*

        if i % 10 == 9:

*# update lr if stabilization happens*

            if (ctr == 990):

                LEARNING\_RATE = 0.01

                print(f'lr updated to {LEARNING\_RATE}')

                optimizer = torch.optim.SGD(

                    model.parameters(), lr=LEARNING\_RATE)

            ctr += 1

            model.eval()

            \_, tr\_pred = tr\_out.max(1)

            n\_samples = tr\_lab.size(0)

            n\_correct = (tr\_pred == tr\_lab).sum().item()

            training\_acc = 100.0 \* n\_correct / n\_samples

            train\_loss = tr\_loss.item()

            val\_total = 0

            val\_correct = 0

*# validation loop*

            for j, val\_data in enumerate(valloader, 0):

                val\_inp, val\_lab = val\_data[0].to(

                    device), val\_data[1].to(device)

                val\_out = model(val\_inp)

                \_, val\_pred = val\_out.max(1)

                val\_total += val\_lab.size(0)

                val\_correct += (val\_pred ==

                                val\_lab).sum().item()

            val\_acc = 100.0 \* val\_correct / val\_total

            train\_loss\_lr.append(train\_loss)

            val\_acc\_lr.append(val\_acc)

tra\_loss\_curve.append(train\_loss\_lr)

val\_acc\_curve.append(val\_acc\_lr)

*# save model*

model\_result = {

    'name': model\_name,

    'loss\_curve\_1': tra\_loss\_curve[0],

    'loss\_curve\_01': tra\_loss\_curve[0],

    'loss\_curve\_001': tra\_loss\_curve[0],

    'val\_acc\_curve\_1': val\_acc\_curve[0],

    'val\_acc\_curve\_01': val\_acc\_curve[0],

    'val\_acc\_curve\_001': val\_acc\_curve[0]

}

*# save model as json file*

with open("Q5\_JSON/Q5\_"+"learning2"+".json", "w") as outfile:

    json.dump(model\_result, outfile)

print("Training Done")

## Code Snippet of 5.1.6.

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

import torchvision

import json

from utils import visualizeWeights

from sklearn.model\_selection import train\_test\_split

*# parameters*

BATCH\_SIZE = 50

NUM\_EPOCHS = 30

NUM\_STEPS = 1

LEARNING\_RATE = 0.1

NUM\_CLASSES = 10

LIMIT = 1000

*# class  definition*

class cnn\_3(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

*# device selection*

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"{device} is available")

print(torch.cuda.is\_available())

print(torch.cuda.current\_device())

print(torch.cuda.device(0))

print(torch.cuda.device\_count())

print(torch.cuda.get\_device\_name(0))

*# transforms*

transform = torchvision.transforms.Compose([

    torchvision.transforms.ToTensor(),

    torchvision.transforms.Normalize(

        (0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),

    torchvision.transforms.Grayscale()

])

*# training set*

trainset = torchvision.datasets.CIFAR10(

    './data', train=True, download=True, transform=transform)

*# splitting training set into training and validation sets*

trainset, valset = train\_test\_split(trainset, test\_size=0.1, random\_state=42)

testset = torchvision.datasets.CIFAR10(

    './data', train=False, transform=transform)

*# loader for training, validation and test sets*

trainloader = torch.utils.data.DataLoader(

    trainset, batch\_size=BATCH\_SIZE, shuffle=True)

valloader = torch.utils.data.DataLoader(

    valset, batch\_size=BATCH\_SIZE, shuffle=False)

testloader = torch.utils.data.DataLoader(

    testset, batch\_size=BATCH\_SIZE, shuffle=False)

classes = ('airplane', 'automobile', 'bird', 'cat',

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

*# loss and accuracy curves*

tra\_loss\_curve = []

val\_acc\_curve = []

*# learning rate curve list*

train\_loss\_lr = []

val\_acc\_lr = []

model = cnn\_3(10)

model = model.to(device)

model\_name = model.\_\_class\_\_.\_\_name\_\_

*# optimizer and loss function*

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING\_RATE)

criterion = torch.nn.CrossEntropyLoss().to(device)

*# ctr for learning rate update*

ctr = 0

*# epoch loop*

for epoch in range(NUM\_EPOCHS):

    print(

        f'Epoch {epoch+1}/{NUM\_EPOCHS}  lr = {LEARNING\_RATE} for {model\_name} model')

*# training loader*

    trainloader = torch.utils.data.DataLoader(

        trainset, batch\_size=BATCH\_SIZE, shuffle=True)

*# training loop*

    for i, tr\_data in enumerate(trainloader, 0):

        model.train()

        tr\_inp, tr\_lab = tr\_data[0].to(device), tr\_data[1].to(device)

*# forward + backward + optimize*

        tr\_out = model(tr\_inp)

        tr\_loss = criterion(tr\_out, tr\_lab)

        optimizer.zero\_grad()

        tr\_loss.backward()

        optimizer.step()

*# record statistics each 10 steps*

        if i % 10 == 9:

*# first update learning rate*

            if (ctr == 990):

                LEARNING\_RATE = 0.01

                print(f'lr updated to {LEARNING\_RATE}')

                optimizer = torch.optim.SGD(

                    model.parameters(), lr=LEARNING\_RATE)

*# second update learning rate*

            if (ctr == 1530):

                LEARNING\_RATE = 0.001

                print(f'lr updated to {LEARNING\_RATE}')

                optimizer = torch.optim.SGD(

                    model.parameters(), lr=LEARNING\_RATE)

            ctr += 1

            model.eval()

            \_, tr\_pred = tr\_out.max(1)

            n\_samples = tr\_lab.size(0)

            n\_correct = (tr\_pred == tr\_lab).sum().item()

            training\_acc = 100.0 \* n\_correct / n\_samples

            train\_loss = tr\_loss.item()

            val\_total = 0

            val\_correct = 0

*# validation loop*

            for j, val\_data in enumerate(valloader, 0):

                val\_inp, val\_lab = val\_data[0].to(

                    device), val\_data[1].to(device)

                val\_out = model(val\_inp)

                \_, val\_pred = val\_out.max(1)

                val\_total += val\_lab.size(0)

                val\_correct += (val\_pred ==

                                val\_lab).sum().item()

            val\_acc = 100.0 \* val\_correct / val\_total

            train\_loss\_lr.append(train\_loss)

            val\_acc\_lr.append(val\_acc)

tra\_loss\_curve.append(train\_loss\_lr)

val\_acc\_curve.append(val\_acc\_lr)

*# save model*

model\_result = {

    'name': model\_name,

    'loss\_curve\_1': tra\_loss\_curve[0],

    'loss\_curve\_01': tra\_loss\_curve[0],

    'loss\_curve\_001': tra\_loss\_curve[0],

    'val\_acc\_curve\_1': val\_acc\_curve[0],

    'val\_acc\_curve\_01': val\_acc\_curve[0],

    'val\_acc\_curve\_001': val\_acc\_curve[0]

}

*# save model as json file*

with open("Q5\_JSON/Q5\_"+"learning3"+".json", "w") as outfile:

    json.dump(model\_result, outfile)

print("Training Done")

## Code Snippet of 5.1.7.

import numpy as np

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

import os

import torch

import torchvision

import json

from utils import visualizeWeights

from sklearn.model\_selection import train\_test\_split

*# parameters*

BATCH\_SIZE = 50

NUM\_EPOCHS = 30

NUM\_STEPS = 1

LEARNING\_RATE = 0.1

NUM\_CLASSES = 10

LIMIT = 1000

*# class  definition*

class cnn\_3(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(cnn\_3, self).\_\_init\_\_()

        self.conv1 = torch.nn.Conv2d(

            in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding='valid')

        self.relu1 = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(

            in\_channels=16, out\_channels=8, kernel\_size=5, stride=1, padding='valid')

        self.relu2 = torch.nn.ReLU()

        self.maxpool1 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv3 = torch.nn.Conv2d(

            in\_channels=8, out\_channels=16, kernel\_size=7, stride=1, padding='valid')

        self.maxpool2 = torch.nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc = torch.nn.Linear(16 \* 3 \* 3, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.maxpool1(x)

        x = self.conv3(x)

        x = self.maxpool2(x)

        x = x.view(BATCH\_SIZE, 16 \* 3 \* 3)

        x = self.fc(x)

        return x

*# device selection*

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"{device} is available")

print(torch.cuda.is\_available())

print(torch.cuda.current\_device())

print(torch.cuda.device(0))

print(torch.cuda.device\_count())

print(torch.cuda.get\_device\_name(0))

*# transform*

transform = torchvision.transforms.Compose([

    torchvision.transforms.ToTensor(),

    torchvision.transforms.Normalize(

        (0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),

    torchvision.transforms.Grayscale()

])

*# training set*

trainset = torchvision.datasets.CIFAR10(

    './data', train=True, download=True, transform=transform)

*# splitting training set into training and validation set*

trainset, valset = train\_test\_split(trainset, test\_size=0.1, random\_state=42)

testset = torchvision.datasets.CIFAR10(

    './data', train=False, transform=transform)

classes = ('airplane', 'automobile', 'bird', 'cat',

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

*# initializing lists*

test\_acc\_curve = []

ctr = 0

*# dataloaders for training, validation and testing*

trainloader = torch.utils.data.DataLoader(

    trainset, batch\_size=BATCH\_SIZE, shuffle=True)

valloader = torch.utils.data.DataLoader(

    valset, batch\_size=BATCH\_SIZE, shuffle=False)

testloader = torch.utils.data.DataLoader(

    testset, batch\_size=BATCH\_SIZE, shuffle=False)

model = cnn\_3(10)

model = model.to(device)

model\_name = model.\_\_class\_\_.\_\_name\_\_

*# optimizer*

opt\_name = "SGD"

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING\_RATE)

*# loss function*

criterion = torch.nn.CrossEntropyLoss().to(device)

*# epoch loop*

for epoch in range(NUM\_EPOCHS):

    print(

        f'Epoch {epoch+1}/{NUM\_EPOCHS}  lr = {LEARNING\_RATE} optimizer = {opt\_name} for {model\_name} model')

    trainloader = torch.utils.data.DataLoader(

        trainset, batch\_size=BATCH\_SIZE, shuffle=True)

*# batch loop*

    for i, tr\_data in enumerate(trainloader, 0):

        model.train()

        tr\_inp, tr\_lab = tr\_data[0].to(device), tr\_data[1].to(device)

*# forward + backward + optimize*

        tr\_out = model(tr\_inp)

        tr\_loss = criterion(tr\_out, tr\_lab)

        optimizer.zero\_grad()

        tr\_loss.backward()

        optimizer.step()

        if i % 10 == 9:

*# learning rate update*

            if (ctr == 990):

                LEARNING\_RATE = 0.01

                print(f'lr updated to {LEARNING\_RATE}')

                optimizer = torch.optim.SGD(

                    model.parameters(), lr=LEARNING\_RATE)

            ctr += 1

*# testing*

with torch.no\_grad():

    model.eval()

    testloader = torch.utils.data.DataLoader(

        testset, batch\_size=BATCH\_SIZE, shuffle=False)

    n\_correct = 0

    n\_samples = 0

*# batch loop*

    for test\_images, test\_labels in testloader:

        test\_images, test\_labels = test\_images.to(

            device), test\_labels.to(device)

        test\_outputs = model(test\_images)

        \_, test\_predicted = test\_outputs.max(1)

        n\_samples += test\_labels.size(0)

        n\_correct += (test\_predicted == test\_labels).sum().item()

    test\_acc = 100.0 \* n\_correct / n\_samples

test\_acc\_curve.append(test\_acc)

*# saving model*

model\_result = {

    'name': model\_name,

    'loss\_curve\_1': test\_acc\_curve[0],

    'loss\_curve\_01': test\_acc\_curve[0],

    'loss\_curve\_001': test\_acc\_curve[0],

    'val\_acc\_curve\_1': test\_acc\_curve[0],

    'val\_acc\_curve\_01': test\_acc\_curve[0],

    'val\_acc\_curve\_001': test\_acc\_curve[0]

}

*# saving model results in json file*

with open("Q5\_JSON/Q5\_"+"learning4"+".json", "w") as outfile:

    json.dump(model\_result, outfile)

print("Training Done")