Erkan Dogan
2166262
2023-04-23

EE449 HW1

1. Basic Concepts

1.1. 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.

1.2. Gradient Computation

From definition of SGD approach:

$$\omega_{k+1} = \omega_k - \gamma * \nabla \mathcal{L}\omega_k$$

Therefore

$$\nabla_\omega \mathcal{L}|_{\omega=\omega_k} = \frac{\omega_k - \omega_{k+1}}{\gamma}$$

1.3. Some Training Parameters and Basic Parameter Calculations

1.3.1. 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.

1.3.2. 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.

1.3.3. 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))

Erkan Dogan
2166262
2023-04-23

1.4. Computing Number of Parameters of ANN Classifiers

1.4.1. Consider an MLP classifier of K hidden units where the size of each hidden unit is H_k for k=1, . . ., K. Derive a formula to compute the number of parameters that the MLP has if the input and output dimensions are D_{in} and D_{out}, respectively.

Input Layer: Input layer has D_{in} units and each unit is connected to the first hidden layer H_1, which results with $D_{in}*H_1$ connections. Also, there is a bias term H_1. In conclusion, parameters formula from first layer is $D_{in}*H_1+H_1$

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

Output Layer: Output layer has D_{out} units, and last hidden layer H_k is connected to output layer. When these connections are made with including the bias D_{out}, the result from output layer is $D_{out}*H_k+D_{out}$

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

$$\#\ of\ parameters = (D_{in} * H_1 + H_1) + \sum_{k=2}^{K} (H_{k-1} * H_k + H_k) + (H_K * D_{out} + D_{out})$$

$$= (D_{in} * H_1) + \sum_{k=1}^{K-1} (H_k * H_{k+1}) + \sum_{k=1}^{K} (H_k) + (H_K * D_{out} + D_{out})$$

1.4.2. Consider a CNN classifier of K convolutional layers where the spatial size of each layer is $H_k×W_k$ and the number of convolutional filters (kernels) of each layer is C_k for k=1, . . ., K. Derive a formula to compute the number of parameters that the CNN has if the input dimension is $H_{in}×W_{in}×C_{in}$.

For each convolutional layer k=1,…,K, the number of parameters can be found by $H_k×W_k×C_k×C_{k-1}+C_k$. For k=1, instead of $C_{k-1=0}$, we can use C_{in}. H_{in} and W_{in} is not in the number of parameters formula since they don't have an impact on learnable parameters.

$$\#\ of\ parameters = (H_1 * W_1 * C_{in} * C_1 + C_1) + \sum_{k=2}^{K} (H_k * W_k * C_{k-1} * C_k + C_k)$$

$$(if\ we\ say\ C_0 = C_{in}) \rightarrow \#\ of\ parameters = \sum_{k=1}^{K} (H_k * W_k * C_{k-1} * C_k + C_k)$$

Erkan Dogan
2166262
2023-04-23

2. Implementing a Convolutional Layer with NumPy

2.1. Experimental Work (Code in Appendix A)

Figure 1 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.

3. Experimenting ANN Architectures

3.1. Experimental Work (Code in Appendix B)

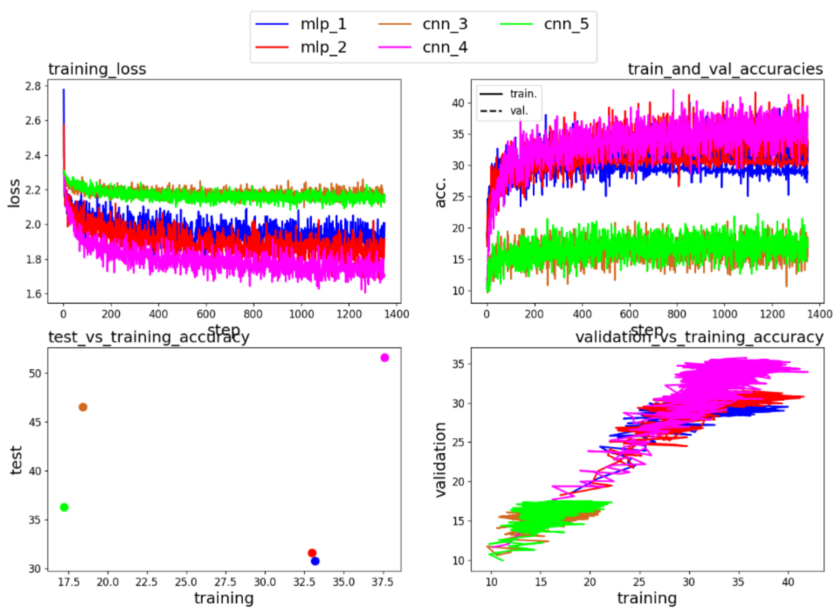


Figure 2 Result Plots of Experimental Work 3.1.

Erkan Dogan
2166262
2023-04-23

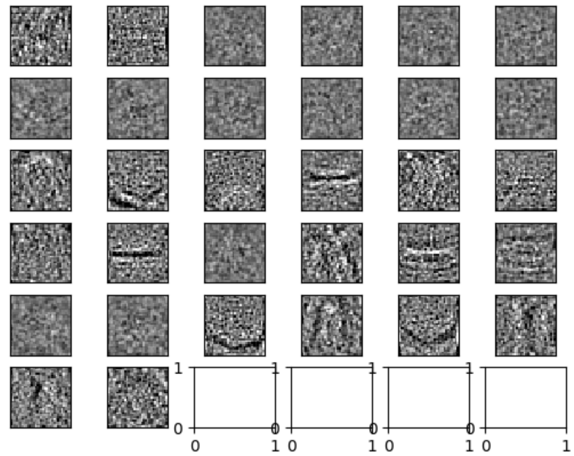


Figure 3 input_weights_mlp_1

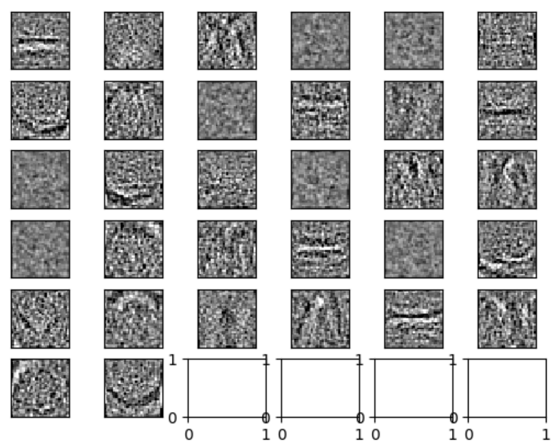


Figure 4 input_weights_mlp_2

Erkan Dogan
2166262
2023-04-23

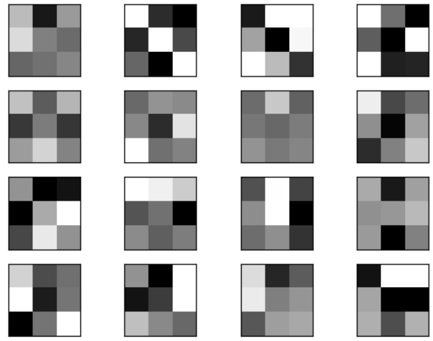


Figure 5 input_weights_cnn_3

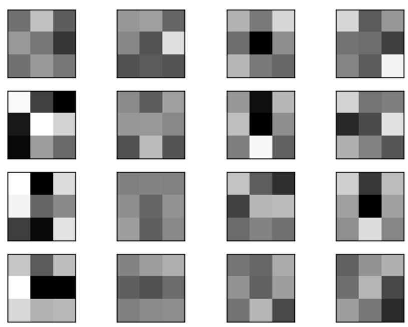


Figure 6 input_weights_cnn_4

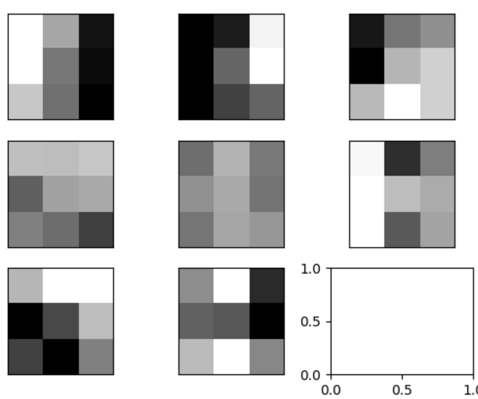


Figure 7 input_weights_cnn_5

3.2. Discussions

3.2.1. 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.

3.2.2. 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.

3.2.3. 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.

3.2.4. 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.

3.2.5. 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.

3.2.6. 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.

3.2.7. 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.

3.2.8. 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.

Erkan Dogan
2166262
2023-04-23

3.2.9. 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.

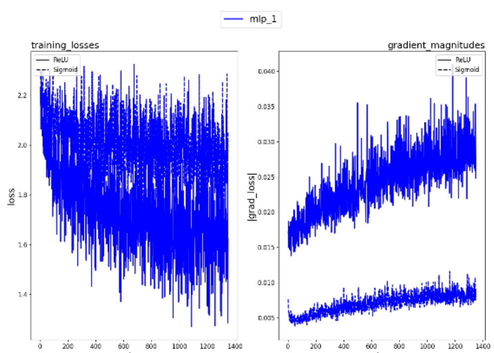
When MLP and CNN models are compared, CNN provides better accuracy, and this is due to the complexity of the dataset we have.

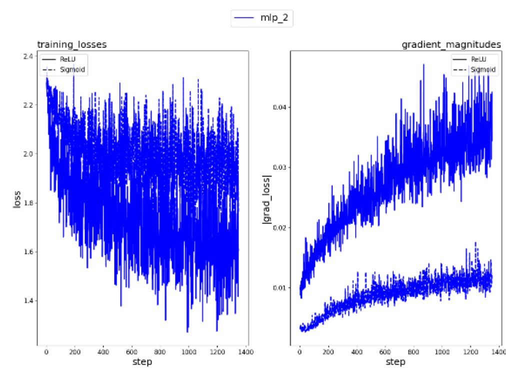
3.2.10. Which architecture would you pick for this classification task? Why?

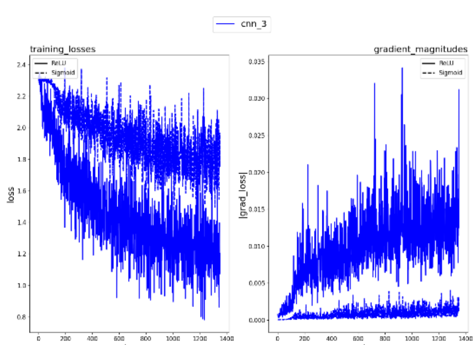
I would choose cnn_4 as it provides better results, and it might have better parameter settings.

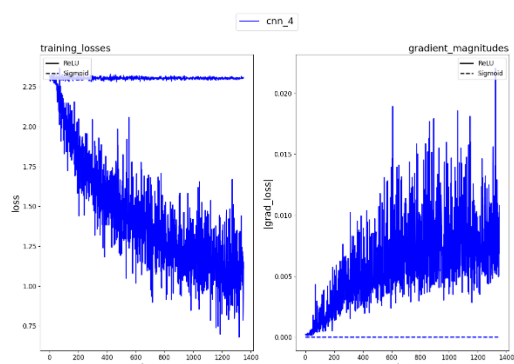
4. Experimenting Activation Functions

4.1. Experimental Work (Code in Appendix C)









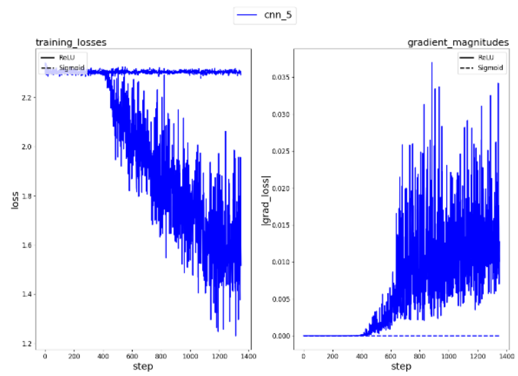


Figure 8 Result Plots from Experimental Work 4.1.

4.2. Discussions

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

Erkan Dogan
2166262
2023-04-23

5. Experimenting Learning Rate

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

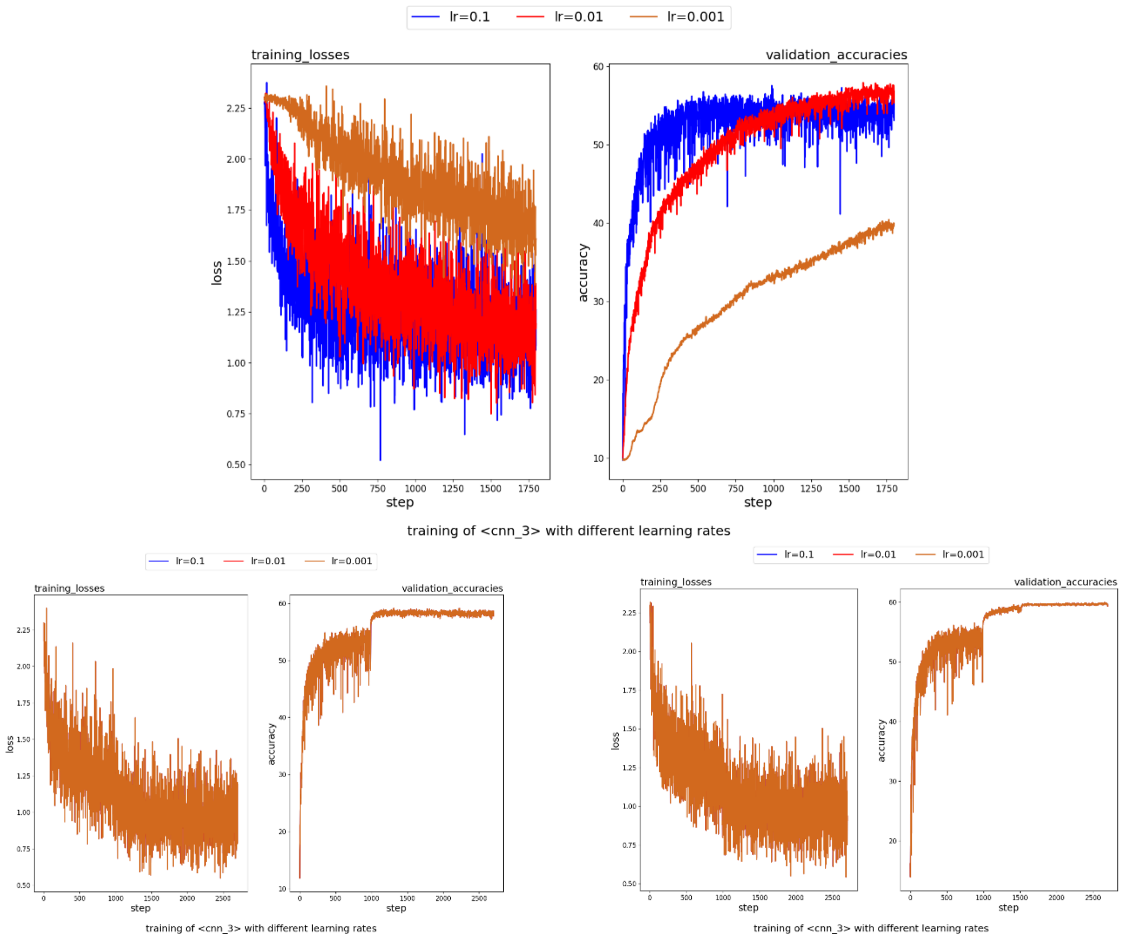


Figure 9 Result Plots from Experimental Work 5.1.

5.2. Discussions

5.2.1. 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.

5.2.2. 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.

5.2.3. 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.

Erkan Dogan
2166262
2023-04-23

5.2.4. 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

A. 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()
```

B. 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)
```

```python
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")
```

C. 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
```

Erkan Dogan
2166262
2023-04-23

```
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')
```

Erkan Dogan
2166262
2023-04-23

```
        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")
```

D. 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")
```

E. 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)
```

```python
    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)
```

Erkan Dogan
2166262
2023-04-23

```
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")
```

F. 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
```

Erkan Dogan
2166262
2023-04-23

```
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")
```

G. 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
```

Erkan Dogan
2166262
2023-04-23

```python
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")
```