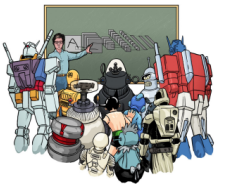



METU EE 449
Computational Intelligence



# Training Artificial Neural Networks

Ozgur Gulsuna

[a]*Middle East Technical University, Electrical and Electronics Engineering, Ankara, Turkey*

**Introduction**

This report explores the basics of ANNs and implement a convolutional layer using the NumPy, and PyTorch libraries. We experiment with different architectures, such as Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN), and analyze the weights of the first layers. We also investigate the impact of different activation functions and learning rates on the performance. Additionally, we explore scheduled learning and how it can improve the performance of our models.

*Keywords:* PyTorch; NumPy; Multi Layer Perceptron; Convolutional Neural Network; Activation Functions; Learning Rate; Scheduled Learning;

## 1. Basic Concepts

### *1.1. Which Function ?*

An ANNs classifier that is trained with cross-entropy loss approximates the conditional probability distribution function. More specifically, for an input data, the output of the classifier is a probability distribution for the classes. The cross-entorpy loss function is a measure between the predicted probability distribution and the true distribution. The form of the loss function is decreasing, smooth and differentiable, which makes it easier to optimize using gradient-based methods. This form is also known as the negative log-like function.

### *1.2. Gradient Computation*

High number of iterations, when the difference between the weights are small, the gradient calculation can be made with basic slope calculation.

$$\gamma \nabla \mathcal{L}_{\omega_k} = \omega_k - \omega_{k+1}$$
$$\nabla_\omega \mathcal{L}\big|_{\omega=\omega_k} = \frac{\omega_k - \omega_{k+1}}{\gamma} \quad \textit{hence,}$$

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

1. The batch refers to a subset of the training data that is used to compute the weights for one iteration. More specifically, the batch size is the number of training samples in a batch. The epoch on the other hand refers to the

*E-mail address:* ozgur.gulsuna@metu.edu.tr

number of times the entire training data is used to update the weights. In training, there are generally multiple epoch iterations where the weights are updated with different batches/subsets of the training data.
2. For the $N$ number of training samples, the number of batches per epoch is $N/B$, where $B$ is the batch size. A little side note that the solution is rounded up to the higher integer if $N/B$ is not an integer.
3. For the optimization iterations, such as SGD, for $E$ number of epochs, the total number of iterations is $E \times N/B$. Again, a practical side note states that the $N/B$ is rounded up to the higher integer.

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

1. Starting from the initial layer of the MLP, we have $D_{in}$ number of input neurons and $H_1$ number of neurons in first hidden layer. Also there are biases associated with each neuron. Therefore, the number of parameters of the each layer is,

$$\begin{aligned} \text{Input Layer} &= D_{in} \times H_1 \ + \ H_1 \\ \text{Hidden Layers} &= H_1 \times H_2 + H_2 \\ &\cdots \\ \text{More Hidden Layers} &= H_{k-1} \times H_k + H_k \\ \text{Output Layer} &= H_k \times D_{out} \ + \ D_{out} \end{aligned}$$

The total sum can be written as, where $K$ is the number of hidden layers.

$$\text{Total Number of Parameters} = D_{in} \times H_1 + \sum_{k=2}^{K} (H_{k-1} \times H_k + H_k) + H_k \times D_{out} + D_{out}$$

2. CNN structure is more complicated. The number of parameters of a CNN layer is calculated as follows:
For the input layer, the number of parameters is,

$$\text{Input Layer} = [(H_{in} - H_1 + 1) \times (W_{in} - W_1 + 1) \times C_{in} \times C_1] + C_1$$

where $H_{in}$ and $W_{in}$ are the height and width of the input image, and $C_{in}$ is the number of channels of the input image. Each input of layer is the output of the previous layer. There exist also biases associated with each neuron added. For the convolutional layers, the number of parameters is calculated as,

$$\text{Convolutional Layer} = [(H_k - H_{k+1} + 1) \times (W_k - W_{k+1} + 1) \times C_k \times C_{k+1}] + C_{k+1}$$

Combination of all layers is,

$$\text{Convolutional Layers} = \sum_{k=1}^{K} [(H_k - H_{k+1} + 1) \times (W_k - W_{k+1} + 1) \times C_k \times C_{k+1}] + C_{k+1}$$

Here all the parameters are summed up. The output is assumed to be the last index of the array. The final equation for the total number of parameters is,

$$\text{Total Parameters} = [(H_k - H_{k+1} + 1) \times (W_k - W_{k+1} + 1) \times C_k \times C_{k+1}] + C_{k+1} + \sum_{k=1}^{K} [(H_k - H_{k+1} + 1) \times (W_k - W_{k+1} + 1) \times C_k \times C_{k+1}] + C_{k+1}$$

## 2. Implementing a Convolutional Layer with NumPy

The section involves implementing conv2d function using NumPy for forward propagation and testing it on a small batch of MNIST dataset. We downloaded and loaded input and kernel files, and created an output image using the

part2Plots function. The implementation code can be found in the appendix named my_conv2d.py. We confirmed the correctness of our implementation by the output image.

### *2.1. Experimental Work*

The generated output for t 1.

Fig. 1. Convolution over the number 8 of the MNIST dataset

### *2.2. Results and Discussion*

1. The Convolutional Neural Networks are important for couple of reasons. First of all, when the input shape of the CNN is selected as 2D, it is well suited image processing. Second, the CNN is able to learn the spatial features of the input image. This also means that the CNN is able to learn and extract the features of the input image without any manual work. CNN's are also able to recognize the features of the input image even if the input image is rotated or scaled. Since the features are extracted from image, partial occlusion of the image affect the performance of the CNN less.
2. The kernel of a Convolutional Layer is essentially a matrix of weights that is convolved (inversely correlated) over with the input data to extract features. The size of the kernel refers to the number of rows and columns in the matrix. It corresponds to the reception of the filter, meaning that higher sizes can extract more complex features.
3. The output image shows that the convolution of pre-presented kernels for the number 8 of the MNIST dataset. Basically each filter is convoled over the images to grasp different meanings. Each column is another kernel with each row is different input image.
4. The numbers in the same column look like each other since they both have a representation of the same number and the same kernel is able to extract the features related to the number 8 other than the specific image.
5. The numbers in the same row look different althougn the input image the same. This is because the kernels are different and they are able to extract different features from the same image.
6. For more specific examples, the third column kernel represents that an 8 has two "islands" of white patches in the middle but the size, shape and location of these pathes differ for each 8 altough all of them represent the same thing in a different manner. Another column such as 6, implies the white track like feature of the number 8. In this sense some features are more dinstinctive than other however when different of these combined make the action work even though they do not seem to represent a clear feature. This is similar to human behaviour as we associate the similar patterns to the general inputs and this is the importance of the convolutaional layers, the features can be learned in a sense.

## 3. Experimenting ANN Architectures

### *3.1. Experimental Work*

This experimental work focuses on testing various Artificial Neural Network (ANN) architectures for a classification task. The models will use adaptive moment estimation (Adam) with default parameters for the optimizer. The

datasets will be preprocessed and split into three sets: training, validation, and testing. The ANN architectures to be tested are ‘mlp 1’, ‘mlp 2’, ‘cnn 3’, ‘cnn 4’, and ‘cnn 5’, each with their specific layers. For each architecture, the ANN will be trained for 15 epochs, and training loss, accuracy, validation accuracy, and test accuracy will be recorded. The best test accuracy and weights of the first layer will be recorded, and a dictionary object will be created and saved for each architecture. Performance comparison plots will be created, and the weights of the first layer of all architectures will be visualized.

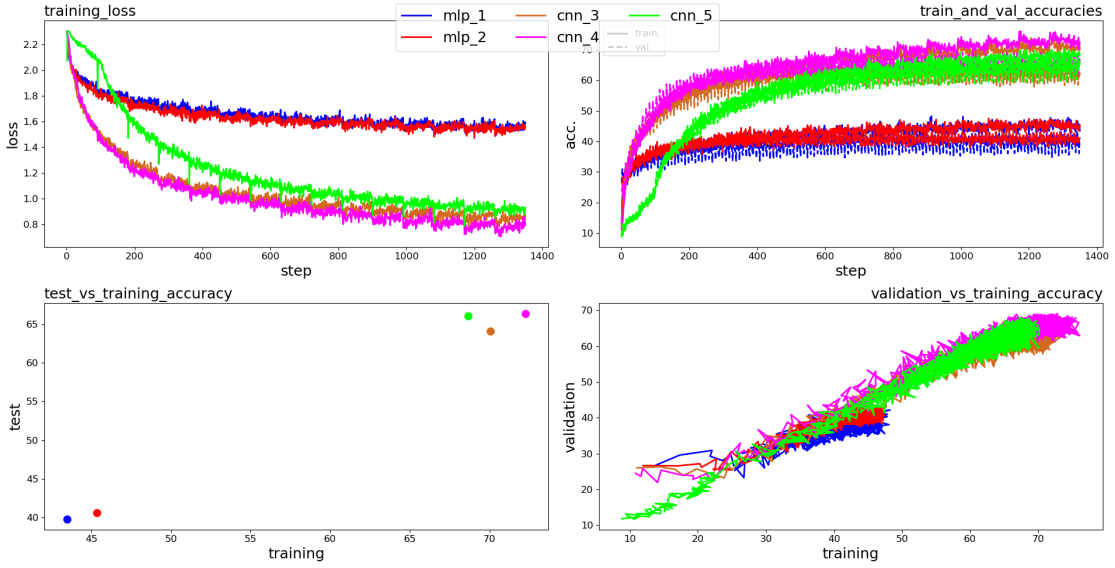
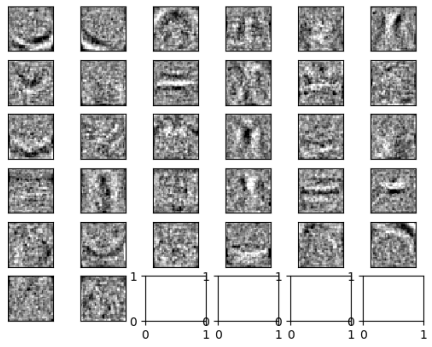


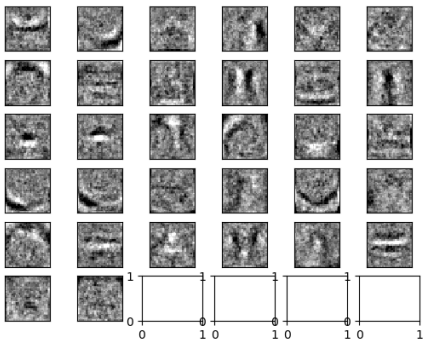

Fig. 2. Performance Comparison Plots for the ANN Architectures

### 3.2. *Results and Discussion*

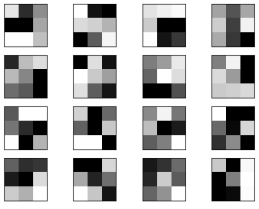
1. Generalization performance refers to the ability of the model to to classify the unseen data. It is used as the ability to recognize patterns and apply this knowledge to the new data.
2. The plots showing the results with the ”test” data show the generalization performance since the test data is not used in the training process and the model is not familiar with the test data. Validation data is also seemed to be a good indicator of the generalization performance. However, altough the validation data is not directly used in training process, it is used to tune the parameters of the model. Therefore, the model is familiar with the validation data in a sense. The first two curves and the last x-y plot give hint about the comparative generalization performance since these show the results with the validation data. The third scatter plot on the other hand is used with the test data, hence have the correct generalization performance. However it only shows the best run hence the variety of the results are a topic of discussion and the plot does not show that information.
3. Copmarative results show that the convolutional architectures perform better than the multi-layer perceptron architectures. This is because the convolutional layers are able to extract spatial features from the data with more grasping ability. The ”mlp_1” and ”mlp_2” are very similar in terms of performance although they have different size of parameters and number of layers. The ”cnn_3” and ”cnn_4” are also very similar again the latter has more layers. The ”cnn_5” is more of a slow learner and could not get to the same level of performance as the ”cnn_4” but with more epochs it is seem to be able to surpass the ”cnn_4” since the gradient of the accuracy is increasing.
4. Higher the number of parameters, it is generally easier for model to learn complex features. However it also means that the model is more prone to overfitting. This is because the model is able to learn the training data in more depth, like its noise characteristics not the required features. Hence, it is not able to generalize well. This is called overfitting, the models with higher parameters have more ”memory” that they can memorize the unwanted characteristics that is specific to the training data. Another aspect is the distribution of the parameters, the



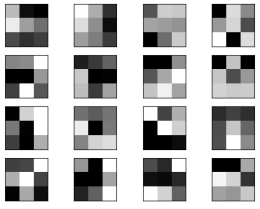
a) mlp_1



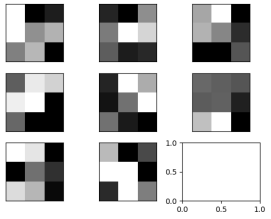
b) mlp_2



c) cnn_3



d) cnn_4



e) cnn_5

convolutaional layers use the parameters more eficiently and make more of the increased number of parameters without easily falling into overfitting trap.

5. The depth of the architecture is also relevelant with the distribution of the parameters, how they organized in an architecture. The models with more depth are able to learn more complex features as well, however they are harder to train in terms of computation. The extremum of depth parameter results in not overfitting but underfitting. This is because the model is not able to learn the features of the data in depth and generalize well.
6. The first layer weights of the MLP structure has high resolution, and can be interpreted as some curves and horizontal lines are perceived. for the CNN structures the weights are more abstract but these architectures have more depth hence a meaning can be found when these are all evaluated together.
7. It is early to say that the units are related with the classes with initial layer weights but horizontal lines can be the plane wing identifier with the tail and body sections are interpreted in the latter layers.
8. At this stage, with the more information on MLP's first layer, it is more interpretable.
9. For each arhchiecture, the depth is increased as explained in previous sections. CNN have an advantage in classification of an image input over the MLP. The higher depth suggests that the CNN is able to learn more complex features and generalize better.
10. I would go with the CNN architecture for image classification task. More specifically "cnn_4" which performs around the same as others with faster training and less parameters as "cnn_5".

## 4. Experimenting Activation Functions

### *4.1. Experimental Work*

This section compares the performance of artificial neural networks (ANNs) using the rectified linear unit (ReLU) and logistic sigmoid activation functions, trained with stochastic gradient descent (SGD) on a constant learning rate of 0.01, 0.0 momentum, and a batch size of 50 samples. For each architecture in section 3.1, two torch.nn.Module objects are created with ReLU and logistic sigmoid activations respectively. The ANNs are trained for 15 epochs,

recording the training loss and magnitude of the loss gradient at every 10 steps. The results are saved as dictionary objects with filenames prefixed with 'part4'. Finally, performance comparison plots are generated.

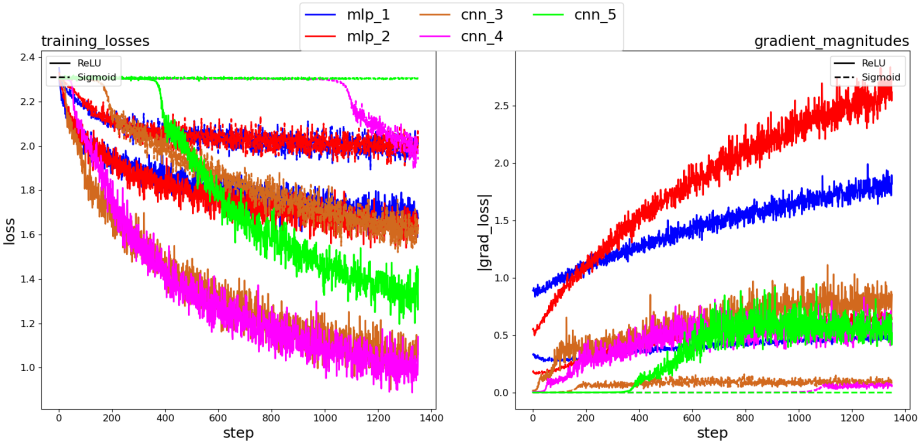



Fig. 3. Performance Comparison Plots for the Different Activation Functions over the Different Architectures.

### *4.2. **Results and Discussion***

1. As the depth increases, the normalized gradient of the loss decreases. Also for the further steps it does not change much, meaning that the learning gets slower. The most complex model "cnn_5" has the lower gradient of the lost for starting epochs. At a point, also loss gradient reduces after a peak value. This also suggests that the training will get slower. The shallow models "mlp_1" and "mlp_2" have relatively high gradients of loss. However, from the minimum loss plot, it is seen that the loss is not decreasing much after a point. This is due model is not deep enough to learn more of the features.
2. Increasing with the depth, gradients start to become harder to calculate. This is because the gradients are calculated by backpropagation and the gradients of the previous layers are needed to calculate the gradients of the current layer. Increasing the complexity at each iteration.
3. Bonus: Since the gradients are calculated from the weights, the unnormalized input can cause gradients to become very small and the learning could take longer. Also for different scales of inputs, the model can give bias to one or more features, which can also reduce the performance of both the training process and the overall system.

## 5. Experimenting Learning Rate

### *5.1. **Experimental Work***

This section explores the effect of varying learning rates in the stochastic gradient descent (SGD) optimization method with a ReLU activation function, 0.0 momentum, and batch size of 50 samples. Three torch.nn.Module objects are created with initial learning rates of 0.1, 0.01, and 0.001, respectively, and trained for 20 epochs. The training loss and validation accuracy are recorded every 10 steps to form loss and accuracy curves. Scheduled learning rate is then explored by training a classifier with a 0.1 learning rate until the epoch step where the validation accuracy stops increasing. The learning rate is then reduced and training continues for another 30 epochs. Finally, the test accuracy of the trained model is compared to the same model trained with Adam in section 3.1.

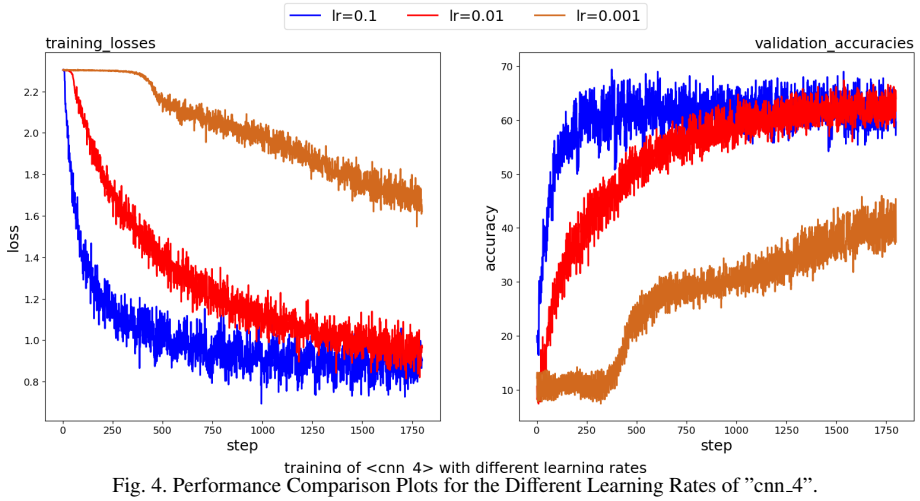



Fig. 4. Performance Comparison Plots for the Different Learning Rates of ”cnn_4”.

## 5.2. *Results and Discussion*

1. Higher learning rates lead to faster convergence, however the model can skip the global minimum. Lower learning rates can lead to a slower convergernce.
2. High learning rate might not get to the minimum point but oscillate around it. Lower has better chance to get to the minimum point.
3. The scheduled learning rate iteratively reduces the learning rate, promising a better convergence with a better step count.
4. With the scheduled learning rate, there is a jump at the epoch 7, which is the point where the learning rate is reduced. The next jump is at epoch 17, however the performance does not increase much after that, suggesting a minimal point is reached. One thing to notice that the training accuracy is higher than the respected model in part 3.1. But the validation accuracy is around the same at 66%.

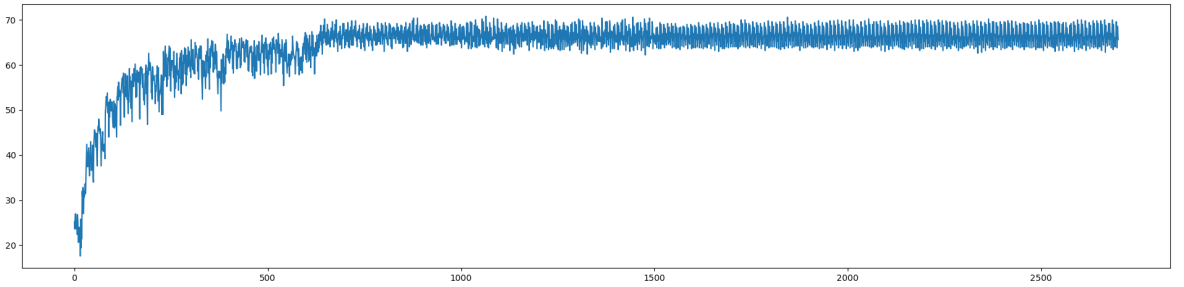


Fig. 5. Validation Accuracy with Scheduled Learning Rate for the ”cnn_4”.

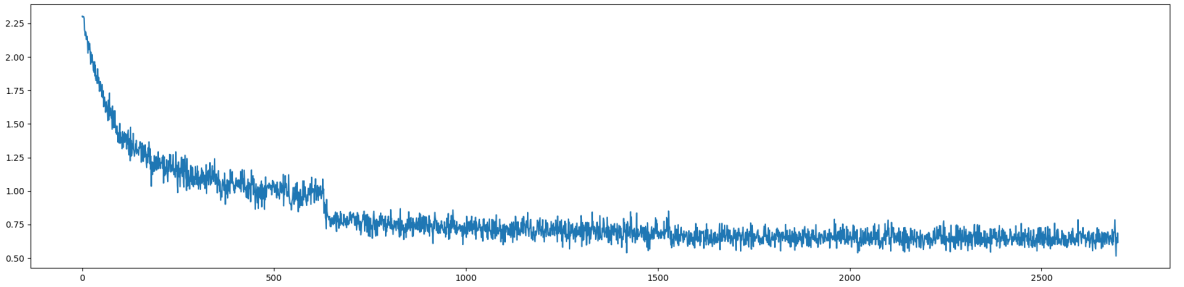


Fig. 6. Training Loss with Scheduled Learning Rate for the ”cnn_4”.

## Appendix A. Python Scripts

```
## HamsterNET: A Convolutional Neural Network that is small and fast, like a hamster running on a wheel.
# Imports ------------------------------------------------------------------------------#
import torch
import torch.nn as nn
import torch.nn.functional as F

import torchvision
import torchvision.transforms as transforms

import numpy as np
import matplotlib.pyplot as plt
from tqdm.notebook import trange, tqdm

import time
import json

# Parameters ---------------------------------------------------------------------------#
validation_ratio = 0.1
batch_size = 50
epoch_size = 15
runs = 5
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
model_name = 'cnn_5'

DISPLAY = False

# Record -------------------------------------------------------------------------------#
save = True
save_path = './HamsNET.pt'
training_loss_record = []
validation_loss_record = []
training_acc_record = []
validation_acc_record = []
test_acc_record = []



# Transformations ----------------------------------------------------------------------#
transform = transforms.Compose([
            torchvision.transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
            # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
            torchvision.transforms.Grayscale()
            ])

# Data ---------------------------------------------------------------------------------#
# test set
test_data = torchvision.datasets.CIFAR10('./data', train = False, download = True, transform = transform)

# training set
train_data_original = torchvision.datasets.CIFAR10('./data', train = True, download = True,transform =
    transform)

# split the training set into training and validation set
train_data, val_data = torch.utils.data.random_split(train_data_original, [int(len(train_data_original)*(1-
    validation_ratio)), int(len(train_data_original)*validation_ratio)])

# lenghts of each set
print('train_data:', len(train_data))
print('val_data:', len(val_data))
print('test_data:', len(test_data))


# Data loader --------------------------------------------------------------------------#
train_generator = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle = True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size = batch_size )
test_generator = torch.utils.data.DataLoader(test_data, batch_size = batch_size )

# Architectures ------------------------------------------------------------------------#
# "mlp_1" is a simple multi-layer perceptron with one hidden layer
class mlp_1(nn.Module):
    def __init__(self, input_size, output_size):
        super(mlp_1,self).__init__()
```

```
            self.input_size = input_size
            self.fc = nn.Sequential(
                nn.Linear(input_size, 32),                          # 1024x32
                nn.ReLU())
            self.prediction_layer = nn.Linear(32, output_size)  # 32x10

        def forward(self, x):
            x = x.view(-1, self.input_size)
            x = self.fc(x)
            x = self.prediction_layer(x)
            return x

    # "mlp_2" is a simple multi-layer perceptron with two hidden layers
    class mlp_2(nn.Module):
        def __init__(self, input_size, output_size):
            super(mlp_2,self).__init__()
            self.input_size = input_size
            self.fc = nn.Sequential(
                nn.Linear(input_size, 32),                          # 1024x32
                nn.ReLU(),
                nn.Linear(32, 64))                                  # 32x64
            self.prediction_layer = nn.Linear(64, output_size)  # 64x10

        def forward(self, x):
            x = x.view(-1, self.input_size)
            x = self.fc(x)
            x = self.prediction_layer(x)
            return x

    # "cnn_3" is a simple convolutional neural network with three convolutional layers
    class cnn_3(nn.Module):
        def __init__(self, output_size):
            super(cnn_3,self).__init__()
            self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)  # 1x32x32 -> 16
                x32x32
            self.relu1 = nn.ReLU()
            self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=5, padding=2)  # 16x32x32 -> 8
                x32x32
            self.relu2 = nn.ReLU()
            self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 8x32x32 -> 8
                x16x16
            self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=7, padding=3)  # 8x16x16 -> 16
                x16x16
            self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 16x16x16 -> 16
                x8x8
            self.prediction_layer = nn.Linear(16 * 8 * 8, output_size)                        # 16x8x8 -> 10

        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(x.size(0), -1)
            x = self.prediction_layer(x)
            return x

    # "cnn_4" is a simple convolutional neural network with four convolutional layers
    class cnn_4(nn.Module):
        def __init__(self, output_size):
            super(cnn_4,self).__init__()
            self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)  # 1x32x32 -> 16
                x32x32
            self.relu1 = nn.ReLU()
            self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x32x32 -> 8
                x32x32
            self.relu2 = nn.ReLU()
            self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5, padding=2)  # 8x32x32 -> 16
                x32x32
            self.relu3 = nn.ReLU()
            self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 16x32x32 -> 16
                x16x16
            self.conv4 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=5, padding=2) # 16x16x16 -> 16
                x16x16
```

```
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                    # 16x16x16 -> 16
            x8x8
        self.prediction_layer = nn.Linear(16 * 8 * 8, output_size)                     # 16x8x8 -> 10

    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.maxpool2(x)
        x = x.view(x.size(0), -1)
        x = self.prediction_layer(x)
        return x

# "cnn_5" is a simple convolutional neural network with six convolutional layers
class cnn_5(nn.Module):
    def __init__(self, output_size):
        super(cnn_5,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, padding=1)   # 1x32x32 -> 8
            x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)  # 8x32x32 -> 16
            x32x32
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x32x32 -> 8
            x32x32
        self.relu3 = nn.ReLU()
        self.conv4 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)  # 8x32x32 -> 16
            x32x32
        self.relu4 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                    # 16x32x32 -> 16
            x16x16
        self.conv5 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, padding=1) # 16x16x16 -> 16
            x16x16
        self.relu5 = nn.ReLU()
        self.conv6 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x16x16 -> 8
            x16x16
        self.relu6 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                    # 8x16x16 -> 8
            x8x8
        self.prediction_layer = nn.Linear(8 * 8 * 8, output_size)                      # 8x8x8 -> 10

    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(x.size(0), -1)
        x = self.prediction_layer(x)
        return x


# Training ------------------------------------------------------------------------#

# initialize your model
if model_name == "mlp_1":
    model = mlp_1(input_size=32*32, output_size=10)
elif model_name == "mlp_2":
    model = mlp_2(input_size=32*32, output_size=10)
elif model_name == "cnn_3":
    model = cnn_3(output_size=10)
elif model_name == "cnn_4":
    model = cnn_4(output_size=10)
```

```
    elif model_name == "cnn_5":
        model = cnn_5(output_size=10)
    else:
        print("Error: model name is not correct!")

    # create loss: use cross entropy loss
    criterion = torch.nn.CrossEntropyLoss()

    # create optimizer
    # optimizer = torch.optim.SGD(model_mlp.parameters(), lr = 0.01, momentum = 0.0)
    optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)

    # Device configuration
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    criterion = criterion.to(device)

    def calculate_accuracy(y_pred, y):
        top_pred = y_pred.argmax(1, keepdim=True)
        correct = top_pred.eq(y.view_as(top_pred)).sum()
        accuracy = correct.float() / y.shape[0]
        return accuracy

    def train(model, iterator, optimizer, criterion, device):
        epoch_loss = 0
        epoch_acc = 0
        step_loss = 0
        step_acc = 0
        model.train()
        i = 0
        for (x, y) in tqdm(iterator, disable=True):
            x = x.to(device)
            y = y.to(device)
            optimizer.zero_grad()
            y_pred = model(x)
            loss = criterion(y_pred, y)
            acc = calculate_accuracy(y_pred, y)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
            epoch_acc += acc.item()
            step_loss += loss.item()
            step_acc += acc.item()
            if i % 10 == 9:
                if DISPLAY is True:                                          # print every 10
                    mini-batches
                    print('[%d, %5d] loss: %.3f' %(epoch + 1, (i+1), step_loss / 10))    # each epoch has
                        5000/50 = 100 steps
                    print('training accuracy: %.2f' % (step_acc*100 / (10)) )           # printed at 10 step
                        intervals
                training_loss_record[run].append(step_loss / 10)                          # save training loss
                    with 10 step intervals
                training_acc_record[run].append(step_acc*100 / 10)                        # save training
                    accuracy with 10 step intervals
                step_loss = 0
                step_acc = 0
            if i % 100 == 99:
                evaluate(model, val_generator, criterion, device, sv=1)
            i += 1
        return epoch_loss / len(iterator), epoch_acc / len(iterator)

    def evaluate(model, iterator, criterion, device,sv=0):
        global best_valid_loss
        epoch_loss = 0
        epoch_acc = 0
        step_loss = 0
        step_acc = 0
        model.eval()
        with torch.no_grad():
            i = 0
            for (x, y) in tqdm(iterator, disable=True):
                x = x.to(device)
                y = y.to(device)
                y_pred = model(x)
                loss = criterion(y_pred, y)
                acc = calculate_accuracy(y_pred, y)
                epoch_loss += loss.item()
```

```
            epoch_acc += acc.item()
            step_loss += loss.item()
            step_acc += acc.item()
            if i % 10 == 9:
                if DISPLAY is True:                                                    # print every 10
                    mini-batches
                    print('[%d, %5d] loss: %.3f' %(epoch + 1, (i+1), step_loss/10))    # each epoch has
                        5000/50 = 100 steps
                    print('validation accuracy: %.2f' % (step_acc*100/10) )           # printed at 10 step
                        intervals
                if sv == 1:
                    validation_loss_record[run].append(step_loss/10 )                  # save
                        validation loss with 10 step intervals
                    validation_acc_record[run].append(step_acc*100/10 )                # save
                        validation accuracy with 10 step intervals
                    if (step_loss/10) < best_valid_loss:
                        best_valid_loss = (step_loss/10)
                        torch.save(model.state_dict(),'./results/trained_models/'+ model_name+'[' +str(run)
                            +'].pt')
                step_loss = 0
                step_acc = 0
            i += 1
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

# Training loop ----------------------------------------------------------------#
torch.save(model.state_dict(), 'empty.pt')

for run in range(runs):

    best_valid_loss = float('inf')
    model.load_state_dict(torch.load('empty.pt'))

    training_acc_record.append([])
    training_loss_record.append([])
    validation_acc_record.append([])
    validation_loss_record.append([])
    test_acc_record.append([])

    for epoch in trange(epoch_size,disable=True):

        start_time = time.monotonic()

        train_loss, train_acc = train(model, train_generator, optimizer, criterion, device)
        valid_loss, valid_acc = evaluate(model, val_generator, criterion, device, sv=0)

        end_time = time.monotonic()

        epoch_mins, epoch_secs = epoch_time(start_time, end_time)
        print(f'+---------------------------------------+')
        print(f'Run: {run+1:02} Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
        print(f'Train Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f} %')
        print(f'Val. Loss: {valid_loss:.3f} | Val. Acc: {valid_acc*100:.2f} %')
        print(f'+---------------------------------------+')

    # Testing ----------------------------------------------------------------------------------#
    # Load the best model in run
    model.load_state_dict(torch.load('./results/trained_models/'+ model_name+'[' +str(run)+'].pt'))

    # Evaluate the model on the test set
    test_loss, test_acc = evaluate(model, test_generator, criterion, device, sv=0)
    print(f'+---------------------------------------+')
    print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
    print(f'+---------------------------------------+')
    test_acc_record[run].append(test_acc*100)

# End of training loop ---------------------------------------------------------------------#

# Load the best model in run ---------------------------------------------------------------#
model.load_state_dict(torch.load('./results/trained_models/'+ model_name+'[' +str(np.array(test_acc_record)
    .argmax())+'].pt'))
```

```
    # Save the first layer weights ------------------------------------------------------#
    # get the weights of first layer [1024x32] as numpy array
    # we used sequential model, so we can access the layers by index: model_mlp.fc[0].weight.data.numpy()
    # we added the .cpu() to move the tensor to cpu memory

    if model_name == 'mlp_1':
        weights = model.fc[0].weight.cpu().data.numpy()
    elif model_name == 'mlp_2':
        weights = model.fc[0].weight.cpu().data.numpy()
    elif model_name == 'cnn_3':
        weights = model.conv1.weight.cpu().data.numpy()
    elif model_name == 'cnn_4':
        weights = model.conv1.weight.cpu().data.numpy()
    elif model_name == 'cnn_5':
        weights = model.conv1.weight.cpu().data.numpy()

    # Save the results ------------------------------------------------------------------#
    with open("./results/["+ model_name +']training_loss_record', "w") as fp:
        json.dump(training_loss_record, fp)
    with open("./results/["+ model_name +']training_acc_record', "w") as fp:
        json.dump(training_acc_record, fp)
    with open("./results/["+ model_name +']validation_loss_record', "w") as fp:
        json.dump(validation_loss_record, fp)
    with open("./results/["+ model_name +']validation_acc_record', "w") as fp:
        json.dump(validation_acc_record, fp)
    with open("./results/["+ model_name +']test_acc_record', "w") as fp:
        json.dump(test_acc_record, fp)
    with open("./results/["+ model_name +']weights.npy', "wb") as fp:
        np.save(fp, weights)


    # FMI : for my information
    # https://stackoverflow.com/questions/72724452/mat1-and-mat2-shapes-cannot-be-multiplied-128x4-and-128x64
```

```
    """-------------------------------------------------------------------------------
2D Convolution with NumPy
-------------------------------------------------------------------------------
Author: Ozgur Gulsuna
Date: 2023-04-14

Description:
This is a simple implementation of 2D convolution with NumPy.

input: [batch size, input_channels, input_height, input_width]
kernel: [output_channels, input_channels, filter_height, filter width]

source: https://www.youtube.com/watch?v=Lakz2MoHy6o
        https://github.com/TheIndependentCode/Neural-Network
-------------------------------------------------------------------------------"""

import numpy as np
import utils

def my_conv2d(input, kernel):
    # kernel transformation done via flip
    for i in range(kernel.shape[0]):
        kernel[i] = np.flip(kernel[i])

    input_height = input.shape[2]
    input_width = input.shape[3]
    input_depth = input.shape[0]
    #print(input_depth)

    kernel_height = kernel.shape[2]
    kernel_width = kernel.shape[3]
    kernel_depth = kernel.shape[0]
    # print(kernel_depth)

    # floor division to get the padding size
    h = kernel_height // 2
    w = kernel_width // 2

    # out = np.zeros((input.shape))
    out = np.zeros((input_depth, kernel_depth,input_height - kernel_height + 1, input_width - kernel_width + 1)
        )
```

```
    # correlate the kernel with the input
    for i in range(input_depth):
        for l in range(kernel_depth):
            for j in range(h,input_height - h):
                for k in range(w,input_width -w):
                    sum = 0
                    for m in range(kernel_height):
                        for n in range(kernel_width):
                            sum += input[i,0, j-h+m, k-w+n] * kernel[l,0, m, n]
                    out[i,l, j-h, k-w] = sum
    return out

# input shape: [batch size, input_channels, input_height, input_width]
input=np.load('./data/samples_8.npy')

# input shape: [output_channels, input_channels, filter_height, filter width]
kernel=np.load('./data/kernel.npy')

# sum = my_conv2d(input, kernel)
out = my_conv2d(input, kernel)

plot = utils.part2Plots(out, nmax=64, save_dir='./out', filename='a')
```

```
import utils
import json
from operator import add
import numpy as np
from matplotlib import pyplot as plt

models = ["mlp_1","mlp_2","cnn_3","cnn_4","cnn_5"]

# results = [[],[],[],[],[]]
results = []
result = {"name": "mpty",
          "loss_curve": [],
          "train_acc_curve": [],
          "val_acc_curve": [],
          "test_acc": 0.0,
          "weights": []}


model_name = "mlp_1"
i = 0
for model_name in models:
    i += 1
    # print(model_name)
    # model_name = str(each)
    # exec("%s = %d" % (model_name +"_training_loss_record", 0))
    training_loss_record = []
    training_acc_record = []
    validation_loss_record = []
    validation_acc_record = []
    test_acc_record = []

    training_loss_record_average=[]
    training_acc_record_average=[]
    validation_loss_record_average=[]
    validation_acc_record_average=[]

    with open("./results/["+ model_name +']training_loss_record', "r") as fp:
        training_loss_record = json.load(fp)
    with open("./results/["+ model_name +']training_acc_record', "r") as fp:
        training_acc_record = json.load(fp)
    with open("./results/["+ model_name +']validation_loss_record', "r") as fp:
        validation_loss_record = json.load(fp)
    with open("./results/["+ model_name +']validation_acc_record', "r") as fp:
        validation_acc_record = json.load(fp)
    with open("./results/["+ model_name +']test_acc_record', "r") as fp:
        test_acc_record = json.load(fp)

    weights=np.load("./results/["+ model_name +']weights.npy')

    # for different runs compute the average loss
    training_loss_record_average = [ sum(x) for x in zip(*training_loss_record) ]
```

```
    training_loss_record_average = [ x/len(training_loss_record) for x in training_loss_record_average ]

    training_acc_record_average = [ sum(x) for x in zip(*training_acc_record) ]
    training_acc_record_average = [ x/len(training_acc_record) for x in training_acc_record_average ]

    validation_loss_record_average = [ sum(x) for x in zip(*validation_loss_record) ]
    validation_loss_record_average = [ x/len(validation_loss_record) for x in validation_loss_record_average ]

    validation_acc_record_average = [ sum(x) for x in zip(*validation_acc_record) ]
    validation_acc_record_average = [ x/len(validation_acc_record) for x in validation_acc_record_average ]

    test_acc_record_best = max(test_acc_record)

    result['name'] = model_name
    result['loss_curve'] = training_loss_record_average
    result['train_acc_curve'] = training_acc_record_average
    result['val_acc_curve'] = validation_acc_record_average
    result['test_acc'] = test_acc_record_best
    result['weights'] = weights.tolist()
    utils.visualizeWeights(weights, save_dir="./out/", filename=model_name+"_weights")


    results.append(result.copy())

# utils.part3Plots(results, save_dir="./results/",filename="aa",show_plot=True)
```

```
# Imports
    ----------------------------------------------------------------------------------------------------------------------

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

import torchvision
import torchvision.transforms as transforms

import numpy as np
import matplotlib.pyplot as plt
from tqdm.notebook import trange, tqdm

import time
import json

# Parameters
    ----------------------------------------------------------------------------------------------------------------------

validation_ratio = 0.1
batch_size = 50
epoch_size = 20
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
model_name = 'cnn_4'

DISPLAY = False

# Record
    ----------------------------------------------------------------------------------------------------------------------

save = True
save_path = './HamsNET.pt'
training_loss_record = []
validation_acc_record = []


# Transformations
    ----------------------------------------------------------------------------------------------------------------------

transform = transforms.Compose([
            torchvision.transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
            # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
            torchvision.transforms.Grayscale()
            ])
```

```
# Data
    ------------------------------------------------------------------------------------------------------------------------------------------

# test set
test_data = torchvision.datasets.CIFAR10('./data', train = False, download = True, transform = transform)

# training set
train_data_original = torchvision.datasets.CIFAR10('./data', train = True, download = True,transform =
    transform)

# split the training set into training and validation set
train_data, val_data = torch.utils.data.random_split(train_data_original, [int(len(train_data_original)*(1-
    validation_ratio)), int(len(train_data_original)*validation_ratio)])

# lenghts of each set
print('train_data:', len(train_data))
print('val_data:', len(val_data))
print('test_data:', len(test_data))

# Data loader
    ------------------------------------------------------------------------------------------------------------------------------------------

train_generator = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle = True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size = batch_size )
test_generator = torch.utils.data.DataLoader(test_data, batch_size = batch_size )

# Architectures
    ------------------------------------------------------------------------------------------------------------------------------------------

# "mlp_1" is a simple multi-layer perceptron with one hidden layer
class mlp_1(nn.Module):
    def __init__(self, input_size, output_size):
        super(mlp_1,self).__init__()
        self.input_size = input_size
        self.fc = nn.Sequential(
            nn.Linear(input_size, 32),                      # 1024x32
            nn.ReLU())
        self.prediction_layer = nn.Linear(32, output_size)  # 32x10

    def forward(self, x):
        x = x.view(-1, self.input_size)
        x = self.fc(x)
        x = self.prediction_layer(x)
        return x

# "mlp_2" is a simple multi-layer perceptron with two hidden layers
class mlp_2(nn.Module):
    def __init__(self, input_size, output_size):
        super(mlp_2,self).__init__()
        self.input_size = input_size
        self.fc = nn.Sequential(
            nn.Linear(input_size, 32),                      # 1024x32
            nn.ReLU(),
            nn.Linear(32, 64))                              # 32x64
        self.prediction_layer = nn.Linear(64, output_size)  # 64x10

    def forward(self, x):
        x = x.view(-1, self.input_size)
        x = self.fc(x)
        x = self.prediction_layer(x)
        return x

# "cnn_3" is a simple convolutional neural network with three convolutional layers
class cnn_3(nn.Module):
    def __init__(self, output_size):
        super(cnn_3,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)  # 1x32x32 -> 16x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=5, padding=2)  # 16x32x32 -> 8x32x32
        self.relu2 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 8x32x32 -> 8x16x16
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=7, padding=3)  # 8x16x16 -> 16x16x16
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 16x16x16 -> 16x8x8
        self.prediction_layer = nn.Linear(16 * 8 * 8, output_size)                        # 16x8x8 -> 10

    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(x.size(0), -1)
        x = self.prediction_layer(x)
        return x

# "cnn_4" is a simple convolutional neural network with four convolutional layers
class cnn_4(nn.Module):
    def __init__(self, output_size):
        super(cnn_4,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)  # 1x32x32 -> 16x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x32x32 -> 8x32x32
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5, padding=2)  # 8x32x32 -> 16x32x32
        self.relu3 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 16x32x32 -> 16
            x16x16
        self.conv4 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=5, padding=2) # 16x16x16 -> 16
            x16x16
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 16x16x16 -> 16x8x8
        self.prediction_layer = nn.Linear(16 * 8 * 8, output_size)                        # 16x8x8 -> 10

    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.maxpool2(x)
        x = x.view(x.size(0), -1)
        x = self.prediction_layer(x)
        return x

# "cnn_5" is a simple convolutional neural network with six convolutional layers
class cnn_5(nn.Module):
    def __init__(self, output_size):
        super(cnn_5,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, padding=1)   # 1x32x32 -> 8x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)  # 8x32x32 -> 16x32x32
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x32x32 -> 8x32x32
        self.relu3 = nn.ReLU()
        self.conv4 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)  # 8x32x32 -> 16x32x32
        self.relu4 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 16x32x32 -> 16
            x16x16
        self.conv5 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, padding=1) # 16x16x16 -> 16
            x16x16
        self.relu5 = nn.ReLU()
        self.conv6 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x16x16 -> 8x16x16
        self.relu6 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 8x16x16 -> 8x8x8
        self.prediction_layer = nn.Linear(8 * 8 * 8, output_size)                         # 8x8x8 -> 10

    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(x.size(0), -1)
        x = self.prediction_layer(x)
        return x


# Training
    ------------------------------------------------------------------------------------------------------------------------------------------------------


# initialize your model
if model_name == "mlp_1":
    model = mlp_1(input_size=32*32, output_size=10)
elif model_name == "mlp_2":
    model = mlp_2(input_size=32*32, output_size=10)
elif model_name == "cnn_3":
    model = cnn_3(output_size=10)
elif model_name == "cnn_4":
    model = cnn_4(output_size=10)
elif model_name == "cnn_5":
    model = cnn_5(output_size=10)
else:
    print("Error: model name is not correct!")

# create loss: use cross entropy loss
criterion = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer1 = torch.optim.SGD(model.parameters(), lr = 0.001, momentum = 0.0)
optimizer2 = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.0)
optimizer3 = torch.optim.SGD(model.parameters(), lr = 0.001, momentum = 0.0)
# optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = criterion.to(device)

def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim=True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    accuracy = correct.float() / y.shape[0]
    return accuracy

def train(model, iterator, optimizer1, criterion, device):
    epoch_loss = 0
    epoch_acc = 0
    step_loss = 0
    step_acc = 0
    model.train()
    i = 0
    for (x, y) in tqdm(iterator, disable=True):
        x = x.to(device)
        y = y.to(device)
        optimizer1.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y)
        acc = calculate_accuracy(y_pred, y)
        loss.backward()
        optimizer1.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
        step_loss += loss.item()
        step_acc += acc.item()
        if i % 10 == 9:
            if DISPLAY is True:                                                    # print every 10 mini-
                batches
                print('[%d, %5d] loss: %.3f' %(epoch + 1, (i+1), step_loss / 10))    # each epoch has 5000/50 =
                    100 steps
                print('training accuracy: %.2f' % (step_acc*100 / (10)) )            # printed at 10 step
                    intervals
            training_loss_record.append(step_loss / 10)                          # save training loss with 10
                step intervals
            step_loss = 0
            step_acc = 0
        if i % 100 == 99:
            evaluate(model, val_generator, criterion, device, sv=1)
```

```
        i += 1
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def evaluate(model, iterator, criterion, device,sv=0):
    global best_valid_loss
    epoch_loss = 0
    epoch_acc = 0
    step_loss = 0
    step_acc = 0
    model.eval()
    with torch.no_grad():
        i = 0
        for (x, y) in tqdm(iterator, disable=True):
            x = x.to(device)
            y = y.to(device)
            y_pred = model(x)
            loss = criterion(y_pred, y)
            acc = calculate_accuracy(y_pred, y)
            epoch_loss += loss.item()
            epoch_acc += acc.item()
            step_loss += loss.item()
            step_acc += acc.item()
            if i % 10 == 9:
                if DISPLAY is True:                                          # print every 10 mini-
                    batches
                    print('[%d,%5d] loss:%.3f' %(epoch + 1, (i+1), step_loss/10))    # each epoch has 5000/50
                        = 100 steps
                    print('validation accuracy:%.2f' % (step_acc*100/10) )          # printed at 10 step
                        intervals
                if sv == 1:
                    validation_acc_record.append(step_acc*100/10 )                   # save validation
                        accuracy with 10 step intervals
                    if (step_loss/10) < best_valid_loss:
                        best_valid_loss = (step_loss/10)
                        torch.save(model.state_dict(),'./results/trained_models/'+ model_name+'[RL001].pt')
                step_loss = 0
                step_acc = 0
            i += 1
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

# Training loop
    ------------------------------------------------------------------------------------------------------------------------


best_valid_loss = float('inf')
for epoch in trange(epoch_size,disable=True):

    start_time = time.monotonic()

    train_loss, train_acc = train(model, train_generator, optimizer1, criterion, device)
    valid_loss, valid_acc = evaluate(model, val_generator, criterion, device, sv=0)

    end_time = time.monotonic()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    print(f'+---------------------------------------+')
    print(f'Epoch:{epoch+1:02}|Epoch Time:{epoch_mins}m{epoch_secs}s')
    print(f'Train Loss:{train_loss:.3f}|Train Acc:{train_acc*100:.2f}%')
    print(f'Val.Loss:{valid_loss:.3f}|Val.Acc:{valid_acc*100:.2f}%')
    print(f'+---------------------------------------+')

# Testing
    ------------------------------------------------------------------------------------------------------------------------


# Load the best model in run
model.load_state_dict(torch.load('./results/trained_models/'+ model_name+'[RL001].pt'))

# Evaluate the model on the test set
test_loss, test_acc = evaluate(model, test_generator, criterion, device, sv=0)
print(f'+---------------------------------------+')
print(f'Test Loss:{test_loss:.3f}|Test Acc:{test_acc*100:.2f}%')
```

```
print(f'+--------------------------------------+')


# Save the results
    ------------------------------------------------------------------------------------------------------------------------

with open("./results/["+ model_name +']RL001_training_loss_record', "w") as fp:
    json.dump(training_loss_record, fp)

with open("./results/["+ model_name +']RL001_validation_acc_record', "w") as fp:
    json.dump(validation_acc_record, fp)



# FMI : for my information
# https://stackoverflow.com/questions/72724452/mat1-and-mat2-shapes-cannot-be-multiplied-128x4-and-128x64
```

```


# Imports
    ------------------------------------------------------------------------------------------------------------------------

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

import torchvision
import torchvision.transforms as transforms

import numpy as np
import matplotlib.pyplot as plt
from tqdm.notebook import trange, tqdm

import time
import json

# Parameters
    ------------------------------------------------------------------------------------------------------------------------

validation_ratio = 0.1
batch_size = 50
epoch_size = 30
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
model_name = 'cnn_4'

DISPLAY = False

# Record
    ------------------------------------------------------------------------------------------------------------------------

save = True
save_path = './HamsNET.pt'
training_loss_record = []
validation_acc_record = []


# Transformations
    ------------------------------------------------------------------------------------------------------------------------

transform = transforms.Compose([
            torchvision.transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
            # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
            torchvision.transforms.Grayscale()
            ])

# Data
    ------------------------------------------------------------------------------------------------------------------------

# test set
test_data = torchvision.datasets.CIFAR10('./data', train = False, download = True, transform = transform)

# training set
train_data_original = torchvision.datasets.CIFAR10('./data', train = True, download = True,transform =
    transform)
```

```
# split the training set into training and validation set
train_data, val_data = torch.utils.data.random_split(train_data_original, [int(len(train_data_original)*(1-
    validation_ratio)), int(len(train_data_original)*validation_ratio)])

# lenghts of each set
print('train_data:', len(train_data))
print('val_data:', len(val_data))
print('test_data:', len(test_data))


# Data loader
# ------------------------------------------------------------------------------------------------------------------

train_generator = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle = True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size = batch_size )
test_generator = torch.utils.data.DataLoader(test_data, batch_size = batch_size )

# Architectures
# ------------------------------------------------------------------------------------------------------------------

# "mlp_1" is a simple multi-layer perceptron with one hidden layer
class mlp_1(nn.Module):
    def __init__(self, input_size, output_size):
        super(mlp_1,self).__init__()
        self.input_size = input_size
        self.fc = nn.Sequential(
            nn.Linear(input_size, 32),                      # 1024x32
            nn.ReLU())
        self.prediction_layer = nn.Linear(32, output_size)  # 32x10

    def forward(self, x):
        x = x.view(-1, self.input_size)
        x = self.fc(x)
        x = self.prediction_layer(x)
        return x

# "mlp_2" is a simple multi-layer perceptron with two hidden layers
class mlp_2(nn.Module):
    def __init__(self, input_size, output_size):
        super(mlp_2,self).__init__()
        self.input_size = input_size
        self.fc = nn.Sequential(
            nn.Linear(input_size, 32),                      # 1024x32
            nn.ReLU(),
            nn.Linear(32, 64))                              # 32x64
        self.prediction_layer = nn.Linear(64, output_size)  # 64x10

    def forward(self, x):
        x = x.view(-1, self.input_size)
        x = self.fc(x)
        x = self.prediction_layer(x)
        return x

# "cnn_3" is a simple convolutional neural network with three convolutional layers
class cnn_3(nn.Module):
    def __init__(self, output_size):
        super(cnn_3,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)  # 1x32x32 -> 16x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=5, padding=2)  # 16x32x32 -> 8x32x32
        self.relu2 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 8x32x32 -> 8x16x16
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=7, padding=3)  # 8x16x16 -> 16x16x16
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 16x16x16 -> 16x8x8
        self.prediction_layer = nn.Linear(16 * 8 * 8, output_size)                        # 16x8x8 -> 10

    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(x.size(0), -1)
        x = self.prediction_layer(x)
```

```
        return x

# "cnn_4" is a simple convolutional neural network with four convolutional layers
class cnn_4(nn.Module):
    def __init__(self, output_size):
        super(cnn_4,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)  # 1x32x32 -> 16x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x32x32 -> 8x32x32
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5, padding=2)  # 8x32x32 -> 16x32x32
        self.relu3 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 16x32x32 -> 16
            x16x16
        self.conv4 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=5, padding=2) # 16x16x16 -> 16
            x16x16
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 16x16x16 -> 16x8x8
        self.prediction_layer = nn.Linear(16 * 8 * 8, output_size)                        # 16x8x8 -> 10

    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.maxpool2(x)
        x = x.view(x.size(0), -1)
        x = self.prediction_layer(x)
        return x

# "cnn_5" is a simple convolutional neural network with six convolutional layers
class cnn_5(nn.Module):
    def __init__(self, output_size):
        super(cnn_5,self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, padding=1)   # 1x32x32 -> 8x32x32
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)  # 8x32x32 -> 16x32x32
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x32x32 -> 8x32x32
        self.relu3 = nn.ReLU()
        self.conv4 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)  # 8x32x32 -> 16x32x32
        self.relu4 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)                                       # 16x32x32 -> 16
            x16x16
        self.conv5 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, padding=1) # 16x16x16 -> 16
            x16x16
        self.relu5 = nn.ReLU()
        self.conv6 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)  # 16x16x16 -> 8x16x16
        self.relu6 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)                                       # 8x16x16 -> 8x8x8
        self.prediction_layer = nn.Linear(8 * 8 * 8, output_size)                         # 8x8x8 -> 10

    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(x.size(0), -1)
        x = self.prediction_layer(x)
        return x
```

```
# Training
    ----------------------------------------------------------------------------------------------------------------------------


# initialize your model
if model_name == "mlp_1":
    model = mlp_1(input_size=32*32, output_size=10)
elif model_name == "mlp_2":
    model = mlp_2(input_size=32*32, output_size=10)
elif model_name == "cnn_3":
    model = cnn_3(output_size=10)
elif model_name == "cnn_4":
    model = cnn_4(output_size=10)
elif model_name == "cnn_5":
    model = cnn_5(output_size=10)
else:
    print("Error: model name is not correct!")

# create loss: use cross entropy loss
criterion = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer1 = torch.optim.SGD(model.parameters(), lr = 0.1, momentum = 0.0)
optimizer2 = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.0)
optimizer3 = torch.optim.SGD(model.parameters(), lr = 0.001, momentum = 0.0)
# optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = criterion.to(device)

def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim=True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    accuracy = correct.float() / y.shape[0]
    return accuracy

def train(model, iterator, optimizer, criterion, device):
    epoch_loss = 0
    epoch_acc = 0
    step_loss = 0
    step_acc = 0
    model.train()
    i = 0
    for (x, y) in tqdm(iterator, disable=True):
        x = x.to(device)
        y = y.to(device)
        optimizer.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y)
        acc = calculate_accuracy(y_pred, y)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
        step_loss += loss.item()
        step_acc += acc.item()
        if i % 10 == 9:
            if DISPLAY is True:                                                    # print every 10 mini-
                batches
                print('[%d, %5d] loss: %.3f' %(epoch + 1, (i+1), step_loss / 10))    # each epoch has 5000/50 =
                    100 steps
                print('training accuracy: %.2f' % (step_acc*100 / (10)) )            # printed at 10 step
                    intervals
            training_loss_record.append(step_loss / 10)                          # save training loss with 10
                step intervals
            step_loss = 0
            step_acc = 0
        if i % 100 == 99:
            evaluate(model, val_generator, criterion, device, sv=1)
        i += 1
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def evaluate(model, iterator, criterion, device,sv=0):
    global best_valid_loss
    epoch_loss = 0
```

```
    epoch_acc = 0
    step_loss = 0
    step_acc = 0
    model.eval()
    with torch.no_grad():
        i = 0
        for (x, y) in tqdm(iterator, disable=True):
            x = x.to(device)
            y = y.to(device)
            y_pred = model(x)
            loss = criterion(y_pred, y)
            acc = calculate_accuracy(y_pred, y)
            epoch_loss += loss.item()
            epoch_acc += acc.item()
            step_loss += loss.item()
            step_acc += acc.item()
            if i % 10 == 9:
                if DISPLAY is True:                                          # print every 10 mini-
                    batches
                    print('[%d, %5d] loss: %.3f' %(epoch + 1, (i+1), step_loss/10))    # each epoch has 5000/50
                        = 100 steps
                    print('validation accuracy: %.2f' % (step_acc*100/10) )          # printed at 10 step
                        intervals
                if sv == 1:
                    validation_acc_record.append(step_acc*100/10 )                    # save validation
                        accuracy with 10 step intervals
                    if (step_loss/10) < best_valid_loss:
                        best_valid_loss = (step_loss/10)
                        torch.save(model.state_dict(),'./results/trained_models/'+ model_name+'[scheduled].pt')
                step_loss = 0
                step_acc = 0
            i += 1
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

# Training loop
    ----------------------------------------------------------------------------------------------------


best_valid_loss = float('inf')
for epoch in trange(epoch_size,disable=True):

    start_time = time.monotonic()

    if epoch < 7:
        train_loss, train_acc = train(model, train_generator, optimizer1, criterion, device)
    elif epoch < 17 and epoch >= 7:
        train_loss, train_acc = train(model, train_generator, optimizer2, criterion, device)
    elif epoch < 31 and epoch >= 17:
        train_loss, train_acc = train(model, train_generator, optimizer3, criterion, device)

    valid_loss, valid_acc = evaluate(model, val_generator, criterion, device, sv=0)

    end_time = time.monotonic()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    print(f'+---------------------------------------+')
    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'Train Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f} %')
    print(f'Val. Loss: {valid_loss:.3f} | Val. Acc: {valid_acc*100:.2f} %')
    print(f'+---------------------------------------+')

# Testing
    ----------------------------------------------------------------------------------------------------


# Load the best model in run
model.load_state_dict(torch.load('./results/trained_models/'+ model_name+'[scheduled].pt'))

# Evaluate the model on the test set
test_loss, test_acc = evaluate(model, test_generator, criterion, device, sv=0)
print(f'+---------------------------------------+')
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
print(f'+-------------------------------------+')


# Save the results
    ------------------------------------------------------------------------------------------------------------------------

with open("./results/["+ model_name +']SC_training_loss_record', "w") as fp:
    json.dump(training_loss_record, fp)

with open("./results/["+ model_name +']SC_validation_acc_record', "w") as fp:
    json.dump(validation_acc_record, fp)

with open("./results/["+ model_name +']SC_test_acc', "w") as fp:
    json.dump(test_acc*100, fp)




# FMI : for my information
# https://stackoverflow.com/questions/72724452/mat1-and-mat2-shapes-cannot-be-multiplied-128x4-and-128x64
```

```
    import utils
import json
from operator import add
import numpy as np
from matplotlib import pyplot as plt

models = ["mlp_1[ReLU]", "mlp_1[Sigmoid]", "mlp_2[ReLU]", "mlp_2[Sigmoid]", "cnn_3[ReLU]", "cnn_3[Sigmoid]", "
    cnn_4[ReLU]", "cnn_4[Sigmoid]", "cnn_5[ReLU]", "cnn_5[Sigmoid]"]

results = []
result = {"name": "mpty",
          "relu_loss_curve": [],
          "sigmoid_loss_curve": [],
          "relu_grad_curve": [],
          "sigmoid_grad_curve": []}

i = 0
for model_name in models:
    i += 1
    training_loss_record = []
    training_grad_record = []

    with open("./results/["+ model_name +']training_loss_record', "r") as fp:
        training_loss_record = json.load(fp)
    with open("./results/["+ model_name +']training_grad_record', "r") as fp:
        training_grad_record = json.load(fp)

    result['name'] = model_name[:5]
    if i % 2 == 1:
        results.append([])
    if model_name[6:13] == "Sigmoid":
        result['sigmoid_loss_curve'] = training_loss_record
        result['sigmoid_grad_curve'] = training_grad_record
        results[int((i-2)/2)]= result.copy()
    elif model_name[6:10] == "ReLU":
        result['relu_loss_curve'] = training_loss_record
        result['relu_grad_curve'] = training_grad_record
        results[int((i-1)/2)]=result.copy()
    print(i)



utils.part4Plots(results, save_dir="./results/", filename='bb', show_plot=True)
```

```
    import utils
import json
from operator import add
import numpy as np
from matplotlib import pyplot as plt

model_name = 'cnn_4'

results = []
```

```
result = {"name": "mpty",
          "loss_curve_1": [],
          "loss_curve_01": [],
          "loss_curve_001": [],
          "val_acc_curve_1": [],
          "val_acc_curve_01": [],
          "val_acc_curve_001": []}

with open("./results/["+ model_name +']RL1_training_loss_record', "r") as fp:
    RL1_training_loss_record = json.load(fp)
with open("./results/["+ model_name +']RL1_validation_acc_record', "r") as fp:
    RL1_validation_acc_record = json.load(fp)
with open("./results/["+ model_name +']RL01_training_loss_record', "r") as fp:
    RL01_training_loss_record = json.load(fp)
with open("./results/["+ model_name +']RL01_validation_acc_record', "r") as fp:
    RL01_validation_acc_record = json.load(fp)
with open("./results/["+ model_name +']RL001_training_loss_record', "r") as fp:
    RL001_training_loss_record = json.load(fp)
with open("./results/["+ model_name +']RL001_validation_acc_record', "r") as fp:
    RL001_validation_acc_record = json.load(fp)

result['name'] = model_name[:5]
result['loss_curve_1'] = RL1_training_loss_record
result['loss_curve_01'] = RL01_training_loss_record
result['loss_curve_001'] = RL001_training_loss_record
result['val_acc_curve_1'] = RL1_validation_acc_record
result['val_acc_curve_01'] = RL01_validation_acc_record
result['val_acc_curve_001'] = RL001_validation_acc_record

utils.part5Plots(result, save_dir="./results/", filename='cc', show_plot=True)
```