1. Basic Concepts

1. Which Function?

An ANNs classifier trained with cross-entropy loss approximates the probability distribution of the target variable based on the input, and it predicts the probability of an input belonging to different classes. The cross-entropy loss measures the difference between the predicted and true probability distribution, and it is defined as the negative sum of the true probabilities multiplied by the logarithm of the predicted probabilities. This loss function is commonly used for classification problems.

The cross-entropy loss function for a classification problem can be written as:

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y})$$

$$\text{where } y \text{ is the true probability distribution}$$
$$\text{and } \hat{y} \text{ stands for predicted probability distribution}$$

The function penalizes the difference between the predicted and true probabilities, with larger penalties for larger differences.

2. Gradient Computation

To compute the gradient of the loss with respect to the weights at step k, the backpropagation algorithm is used. The gradient of the loss with respect to the output of the neural network is computed first, and then the gradient with respect to the weights is computed using the chain rule. With the SGD *(Stochastic Gradient Descent)* approach, the weight update rule is used to express the gradient in terms of the learning rate, and the difference between the weights at iterations k and k+1. Therefore, the gradient of the loss with respect to the weights at step k can be obtained by multiplying the gradient of the loss with respect to the output with the derivative of the output with respect to the weights and scaling the result by the learning rate and the difference between the weights at iterations k and k+1.

The gradient with respect to the weights can be computed using the chain rule as:

$$\frac{\partial L}{\partial \omega} = \frac{\partial L}{\partial x} \frac{\partial x}{\partial \omega}$$

Using the SGD approach, the weight update rule at each iteration k is given by:

$$\omega_{k+1} = \omega_k - \Delta\, \omega_k$$

$$\omega_{k+1} = \omega_k - \{ \gamma \frac{\partial L}{\partial \omega} \}$$

$$\omega_{k+1} = \omega_k - \{ \gamma \frac{\partial L}{\partial x} \frac{\partial x}{\partial \omega} \}$$

$$\omega_{k+1} - \omega_k = -\gamma \frac{\partial L}{\partial x} \frac{\partial x}{\partial \omega}$$

$$\frac{(\omega_{k+1} - \omega_k)}{-\gamma} = \frac{\partial L}{\partial x} \frac{\partial x}{\partial \omega}$$

$$\frac{(\omega_k - \omega_{k+1})}{-\gamma} = \frac{\partial L}{\partial x} \frac{\partial x}{\partial \omega} = \frac{\partial L}{\partial \omega} \Big|_{\omega=\omega_k}$$

3. Some Training Parameters and Basic Parameter Calculations

1. In machine learning, training a neural network involves iterating through the dataset multiple times, updating the weights of the network to minimize the loss function. The two key concepts in MLP training are batches and epochs.

   A **batch** is a subset of the training data that is used to compute the gradient of the weights during the optimization process. The batch size determines how many samples are processed in each iteration.

   For example, if the training dataset contains 10,000 samples and we set the batch size to 32, then the optimizer will update the weights every 32 samples. The batch size is a hyperparameter that needs to be tuned for optimal results.

   An **epoch** is one complete pass through the entire training dataset. During an epoch, the optimizer processes each batch of training data once.

   For example, if we have a training dataset of 10,000 samples and a batch size of 32, then one epoch would involve 313 iterations (i.e., 10,000/32). After one epoch, the MLP updates its weights based on the gradient computed for each batch of data. Typically, the number of epochs is a hyperparameter that is set based on the size of the dataset and the complexity of the MLP.

2. If the dataset has N samples and the batch size is B, then the number of batches per epoch can be calculated as:

$$number\ of\ batches\ per\ epoch = ceil\ (\frac{N}{B})$$

"Ceil" is the ceiling function that rounds up to the nearest integer. The last batch in the epoch may contain fewer than B samples.

3. If we want to train our ANN for E epochs with a batch size of B, then the total number of stochastic gradient descent (SGD) iterations can be calculated as:

$$number\ of\ SGD\ itereations = E \ \times ceil\,(\frac{N}{B})$$

4. Computing Number of Parameters of ANN Classifiers

1. Mentioned ANN Classifier, that is demonstrated in *Figure 1*, has one input layer with D_{in} number of nodes, K hidden layers with H_k number of nodes where k = 0, 1,…,K and one output layer with D_{out} number of nodes.

   To calculate the number of parameters between the input layer and the first hidden layer, number of weights and the number of biases must be found. Considering that there is also a weight for the bias term, there are $(D_{in} + 1)$ number of weights. To find the total, that number is multiplied with the number of nodes in the next layer. Thus, number of weights is $H_1\ (D_{in} + 1)$. There is one bias term for layer. Hence, there are $H_1(D_{in} + 1) + 1$ parameters between the input layer and the first hidden layer. Same logic applies for all layers. There are K hidden layers with K biases. In total, there are K + 1 biases. Number of parameters are written for different layers in *Table 1*.

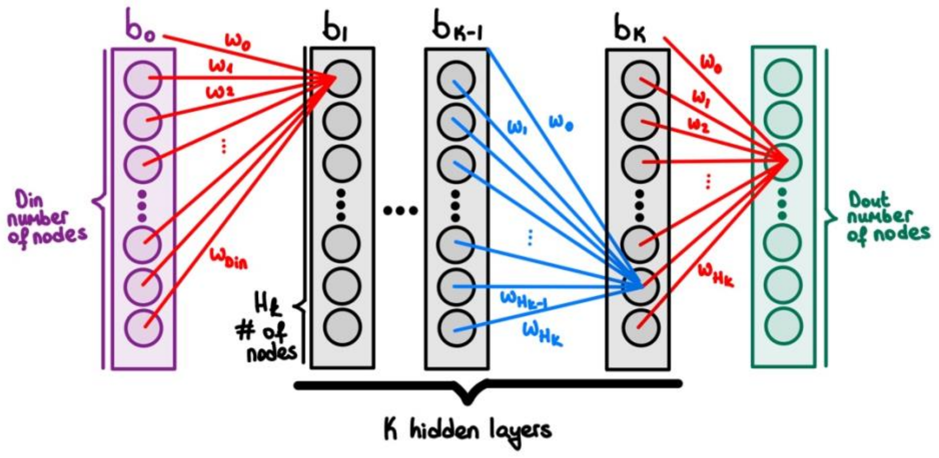


FIGURE 1. MLP CLASSIFIER LAYER DEMONSTRATION.

Table 1: Number of parameters in the MLP Classifier.

	Between the input and the first hidden layer	Between two hidden layers	Between the last hidden layer and the output layer
Number of weights	$(D_{in}+1)\,H_1$	$(H_{k-1}+1)\,H_k$	$(H_k+1)\,D_{out}$
Number of biases	1	1	1

$$Total\ number\ of\ parameters$$
$$= (D_{in}+1)\,H_1 + 1 + \sum_{k=2}^{K}((H_{k-1}+1)\,H_k + 1)\ + (H_k+1)\,D_{out} + 1$$
$$= \ (D_{in}+1)\,H_1 + \sum_{k=2}^{K}((H_{k-1}+1)\,H_k)\ + K + 1$$

2. Second case is similar to the first one. CNN Classifier is demonstrated in *Figure 2*. Number of kernels is determined as C_k as "k" represents the hidden layer number. There will be one bias for each kernel. Therefore, the C_k terms provide us with the number of biases at each layer. Moreover, number of weights will be equal to the dimensions of the next layer (output layer) multiplied with the C term of the input layer. For example, number of weights at the first convolutional layer is $H_1 W_1 C_{in} C_1$ and the number of biases is C_1. Number of parameters are written for different layers in *Table 2* for the CNN Classifier.

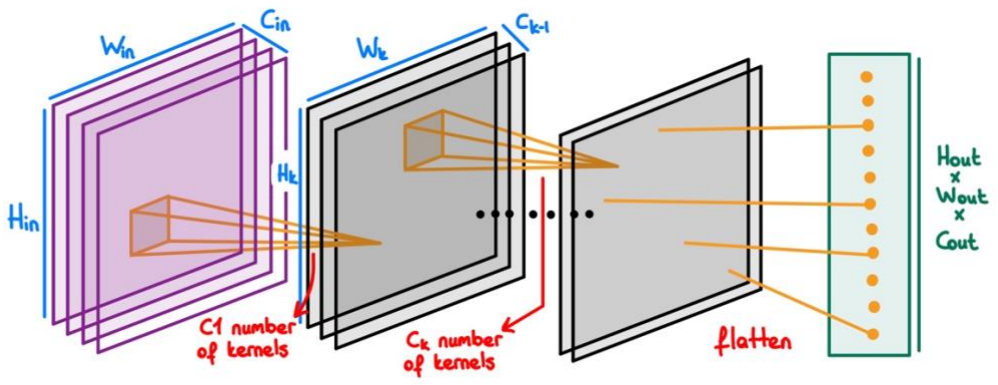


FIGURE 2. CNN CLASSIFIER LAYER DEMONSTRATION.

Table 2: Number of parameters in the CNN Classifier.

	In the first convolutional layer	In each subsequent convolutional layer
Number of weights	$H_1 W_1 C_{in} C_1$	$H_k W_k C_{k-1} C_k$
Number of biases	C_1	C_k

$$\begin{aligned} &Total\ number\ of\ parameters \\ &= H_1 W_1 C_{in} C_1 + C_1 + \sum_{k=2}^{K} (H_k W_k C_{k-1} C_k + C_k) \end{aligned}$$

Investing both models, it can be seen that the calculation of the parameters are done in similar way. Input sizes differ, and kernel size is introduced to process the input in a more comprehensive way. Moreover, processing the input with more than one dimension will definitely be easier when the input is inspected in smaller pieces.

2. Implementing a Convolutional Layer with NumPy

1. Experimental Work

My student number is 2444081. My conv2d function does a 2D Convolution on the input file “samples_1.npy” with the kernel provided in “kernel.npy”. I obtained the following output in *Figure 3* with the conv2d function that I have written.

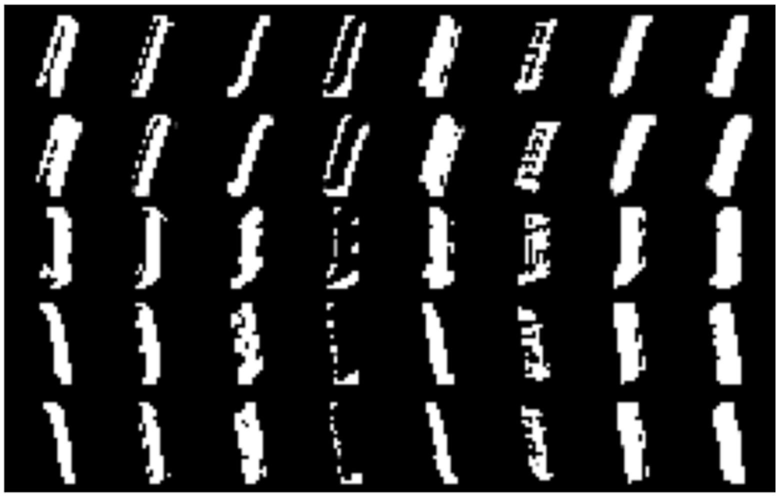


FIGURE 3. OUTPUT FROM THE HAND-WRITTEN CONV2D FUNCTION.

As the input is normalized, following output in *Figure 4* is observed as the output.

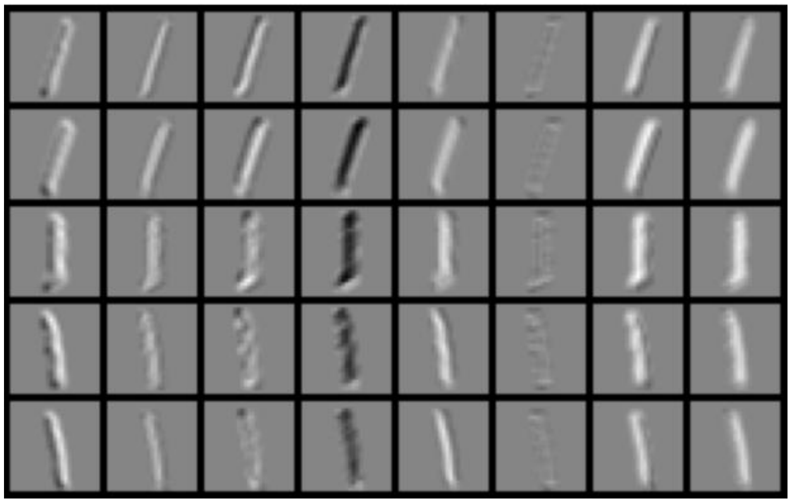


FIGURE 4. OUTPUT FROM THE CONV2D FUNCTION WHEN THE OUTPUT IS NORMALIZED.

Inspecting the outputs in Figure 3 and 4, it can be deducted that image obtained is clearer when the output is normalized. The outputs contain various images of the number 1.

2. Discussions

1. Convolutional Neural Networks (CNNs) are significant because they have the power to learn (not memorize) relevant features from unprocessed data such as images and use those features to make precise predictions and classifications.

   CNNs are especially useful in image processing tasks due to their capability to detect visual patterns and spatial relationships within an image. At the first layers, minimal features that objects are made of, are detected such as lines, dots, corners… As these small patterns are detected, combinations of these minimal features are sought, for images of objects consist of these. For example, in order to detect a face, the classifier must be able to detect eyes, nose, mouth, ears… Object detection model seeks in fact circles, triangles and rectangles in different colors. And these shapes are made of lines and corners. Hence, small features are detected initially, then combined.

   Unlike conventional neural networks, CNNs have convolutional layers that filter the input data and lower its dimensionality while retaining essential features, making them more efficient and effective at image processing tasks. Furthermore, CNNs use a pooling layer that further reduces the spatial dimensions of the input data to minimize overfitting and improve generalization performance. The resulting feature maps are then flattened and fed into fully connected layers that can be employed for classification, regression, or other tasks.

Overall, CNNs are an important tool in image processing for they are able to learn intricate visual features directly from unprocessed data and use those features to make accurate predictions or classifications. They are widely used in a range of applications such as object recognition, image segmentation, and image synthesis.

2. In Convolutional Layers of a Convolutional Neural Network, the kernel is a small matrix that slides over the input data, performs a dot product at each location, and generates a feature map that represents the presence of certain features or patterns in the input data. The size of the kernel corresponds to the receptive field of the Convolutional Layer, which is the area of the input data that each neuron in the feature map can detect. A larger kernel can detect larger patterns but increases the number of parameters and computational cost, while a smaller kernel can detect smaller patterns but may miss larger features. The size of the output feature map is determined by the kernel size and stride, and the depth of the feature map is determined by the number of kernels.

3. In 2D convolution using a library like Numpy, the kernel moves across the input image and computes the dot product of corresponding pixels and kernel elements at each position. This generates a new pixel value in the output feature map that is smaller than the input image due to the kernel size and stride used. The output feature map represents the detected features or patterns in the input image and can contain negative or positive values depending on the kernel and input image. To improve the expressive power of the network, the output image may be passed through an activation function like ReLU to introduce non-linearity.

4. Each column of numbers in a CNN output corresponds to the activations of identical neurons in the same layer. Hence, the outputs in the same column look alike since similar features are extracted from the input and put in the same column. Although they are not exactly identical, shapes and shadings resemble each other.

5. Each row of numbers in a CNN output corresponds to the activations of different neurons in the same layer. Although they belong to the same input image, they may not look alike since each neuron has a specific role in detecting different features or patterns. As a result, the activation values in the same row may vary depending on the features and patterns detected by the corresponding neurons, leading to a diverse range of values within the row.

6. Convolution operation is used widely in computational intelligence. Although they are not used in this question, with the activation functions and the cascaded convolutional layers, classifiers are able to extract features which is the essence of object detection. In this question, the impact of the convolution operation is observed with a sample image and a kernel. Effects of convolution are visible in the obtained image (output).

3. Experimenting ANN Architectures

1. Experimental Work

In this part, 5 different ANN architectures are implemented. Two of these architectures are MLPs, while the other three are CNNs. Number and features of layers and the activation function in between differ. Thus, they yield different results. Constructing these architectures, the models are created, and each model is trained ten times using the CIFAR-10 dataset, each training consists of 15 epochs with batch size of 50. This dataset consists of 50,000 images each belonging 10 different class of objects (airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships, and trucks). Each model has the same final layer. A fully connected layer with 10 neurons which provides the probability of the input image belonging to each class.

While training the models, training loss, training accuracy and validation accuracy are recorded at each 10 steps. After training the models, average of training loss, training accuracy and validation accuracy are taken to have a clearer sense on the performance of the models. Then, the resulting data are saved as dictionary objects in Pickle files. The performance of the models are determined by the average of their training loss, training accuracy and validation accuracy. At the final stage, models that provides the best test accuracies are inspected and the results are plotted which can be seen in *Figure 5*, using the plotting functions provided in utils.py file.

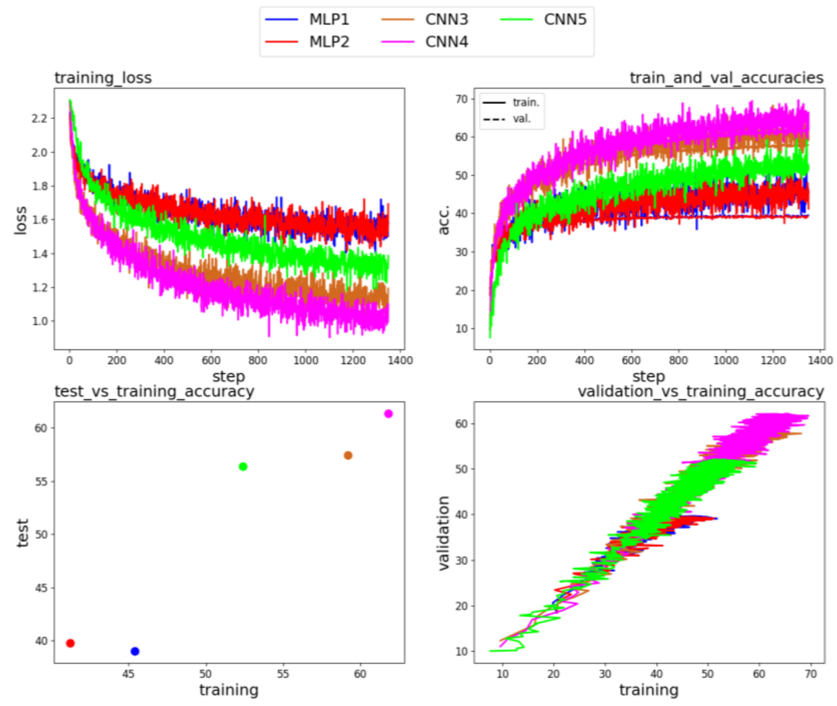


FIGURE 5. RESULTS OF THE 5 MODELS WITH DIFFERENT ARCHITECTURES TRAINED WITH THE CIFAR-10 DATASET.

Inspecting the plots obtained, it can be deducted that CNN4 provides the best performance with lower training losses ad higher training and validation accuracies.

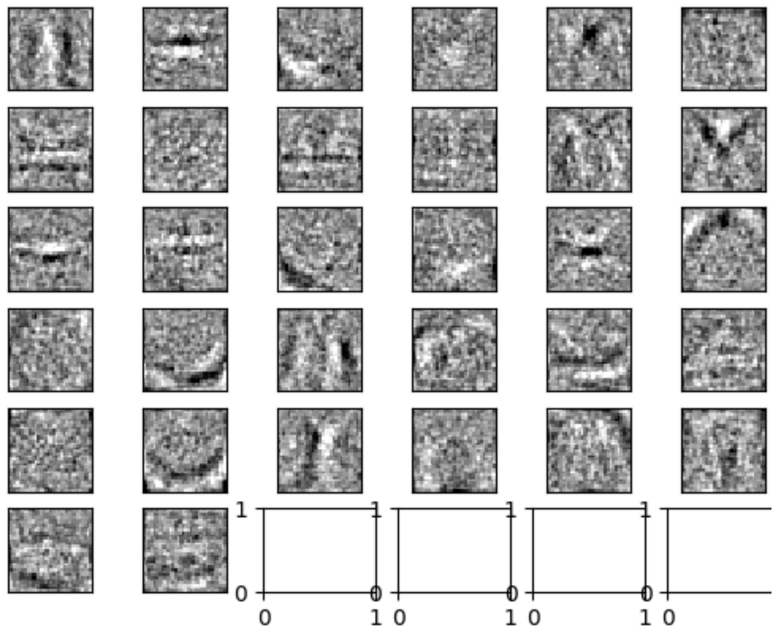


FIGURE 6. VISUALIZED WEIGHTS FOR MLP1.

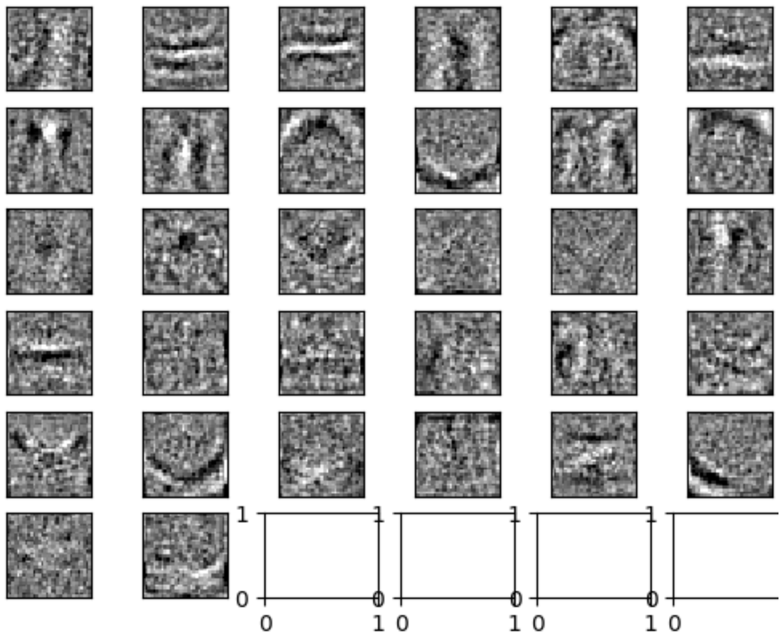


FIGURE 7. VISUALIZED WEIGHTS FOR MLP2.

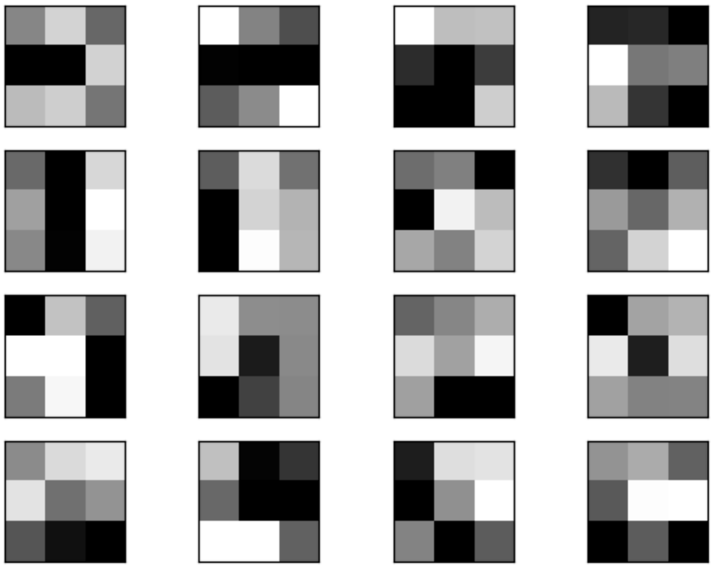


FIGURE 8. VISUALIZED WEIGHTS FOR CNN3.

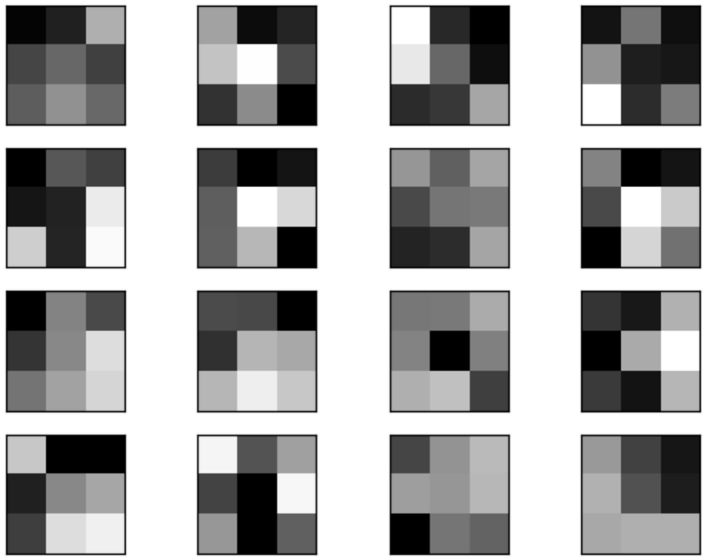


FIGURE 9. VISUALIZED WEIGHTS FOR CNN4.

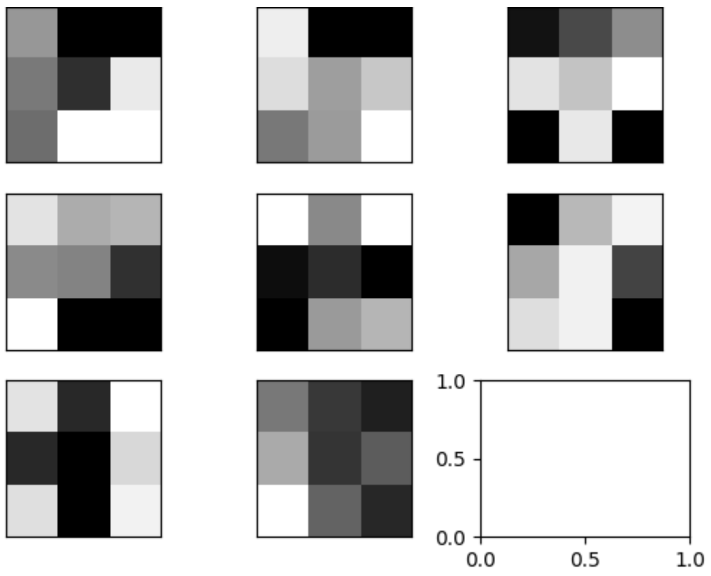


FIGURE 10. VISUALIZED WEIGHTS FOR CNN5.

2. Discussions

1. When constructing different Artificial Neural Network Architectures for a classification task such as Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs), the generalization performance of a classifier depends on its ability to accurately detect and recognize new and unseen data (data that is not pre-processed or used before).

   For a classifier training loss must decrease as the training process continues. Moreover, the dataset must not be memorized since that way the model does not learn and the classifier performs poorly with the new data provided as input to it. To prevent this, the datasets are split into different sections namely, train set, validation set and test set. While the model is trained with the train set, the performance of the model is supervised, inspecting its classification performance on the validation set. A decreasing training loss means that the model is learning. However, if the validation accuracy is not increasing continuously and converging to a value while the training continues with the updated weights, then this means that the model is memorizing the input provided to it. This phenomenon is known as overfitting. To have a classifier that detects objects at a high performance, overfit must be prevented.

2. As mentioned earlier, a decreasing training loss means that the model used is able to predict the images better and better as the training process continues. In other words, lower the training loss lower the error. Test accuracy shows how well the model is doing with the data provided to it from the training set. Validation accuracy indicates how good is the model at generalizing new data that is isolated from the model during the training process.

Training loss vs. step plot, shows us the models are getting better at predicting and classifying the objects in the images from the train set, meaning that the error is decreasing. Second and fourth plots demonstrate what the trend of training and validation accuracy, as the training continues. It is desired to observe an increasing validation accuracy as the training accuracy increases. Otherwise, the model may be memorizing the data in the training set and will do poorly on classifying new data (data from outside the train set). Lastly, third plot shows us how well the model is classifying the data from the train set and the test set. A trained model with a low training accuracy will also have low test accuracy. However, a high training accuracy does not always mean that the test accuracy will be high as well.

3. In general, Convolutional Neural Networks (CNN3, CNN4 and CNN5) performed better comparing to the Multi-Layer Perceptrons (MLP1 and MLP2). Within the architectures of CNNs, there were more layers. Moreover, more activation functions (ReLU) and pooling layers (MaxPool) are added between the convolutional layers to increase their performance. While a simple matrix multiplication operation is done within the MLPs, convolution operations in the CNN architectures were better at extracting features that make up the objects.

4. Number of parameters, i.e., weights and biases, are increased as the number of layers is increased. CNN models had more layers and activation functions in between. As it can be seen from the plots in *Figure 5*, the models with more parameters do better and provide better results. CNN4 had more layers than CNN3 and yielded more reliable results. However, as the parameters increased and new layers are added as constructing CNN5, it is observed that CNN5 had lower performance than CNN4. Meaning that, increasing layers, adding more and more parameters does not always lead to a better trained model. Number of parameters should be calibrated carefully.

5. Depth of the architecture (number of layers) harms the performance of the models after some point. While a model with minimum layers will not be sufficient to classify the objects as the model will not be able to decrease its error rate during training, a model with excessive amount of layers lead to problems such as vanishing gradient problem (gradient of the loss function will be close to zero), difficulty in hyperparameter tuning (it will be more difficult to find the right learning rate), increased complexity and computation (it will be more difficult to train the model), and overfitting (memorizing the dataset).

6. Weights of the first layer from the MLP models are more interpretable than the visual weights obtained from the CNNs, since the features extracted are clearer. However, it is not easy to put meaning into the visualized weights due to the fact that the first layers extract small features.

7. As mentioned earlier, first layer weights will not provide sufficient visualization to recognize the features of the output classes since the weights are obtained from the

first layers of each model. Features of the classes will be more observable in the subsequent layers. The features will be most clear at the last layer (prediction layer).

8. The weights in the first layer of an MLP are directly linked to the input features, making it easier to understand their significance or role in the model's output. In contrast, the weights in the first layer of a CNN are filters that detect features in different parts of the input image, which may not have a direct interpretation in terms of the original input features. Furthermore, the filters in the first layer of a CNN generally represent low-level features such as edges or corners, which may not be as readily interpretable as the original input features. Therefore, the weights in the first layer of an MLP are more interpretable than those in a CNN because they are directly linked to the input features, whereas the filters in the first layer of a CNN are typically low-level image features that are more challenging to interpret in terms of the original input features. Hence, the weights in the first layer of an MLP are more interpretable than the ones extracted from CNNs.

9. MLP1 and MLP2 have similar architecture, while CNN3, CNN4 and CNN5 are akin to each other, as expected. CNN architectures have multiple convolutional layers and MLPs utilize fully connected layers. As can be seen from the plots in *Figure 5*, CNNs provide more performance on the classification task. Although their capabilities differ, each CNN model does better than each MLP model. This leads to a conclusion, that the convolution operation plays a crucial role in artificial intelligence.

10. If the number of epochs at each training is not changeable, CNN4 is the most reasonable choice for an architecture. Training, validation, and test accuracy is greater for CNN4, providing a better classification performance.

4. Experimenting Activation Functions

1. Experimental Work

In the fourth part, the structure of the architectures we have built in the third part are altered. The impact of the activation functions is investigated for the 5 ANN models, namely MLP1, MLP2, CNN3, CNN4 and CNN5. Instead of ReLU (Rectified Linear Unit), Sigmoid activation is used as the activation functions. Models trained twice for two functions, .i.e., MLP_R and MLP_S, two models with activation functions of ReLU and Sigmoid, respectively. Additionally, the optimizer is change from Adam to SGD.

Making a comparison between the two activation functions, training loss and gradient magnitudes are inspected. The results of the trainings are plotted using part4Plots function which was located in the utils.py file. Plots for the training losses and gradient magnitudes are visible in *Figure 11*.

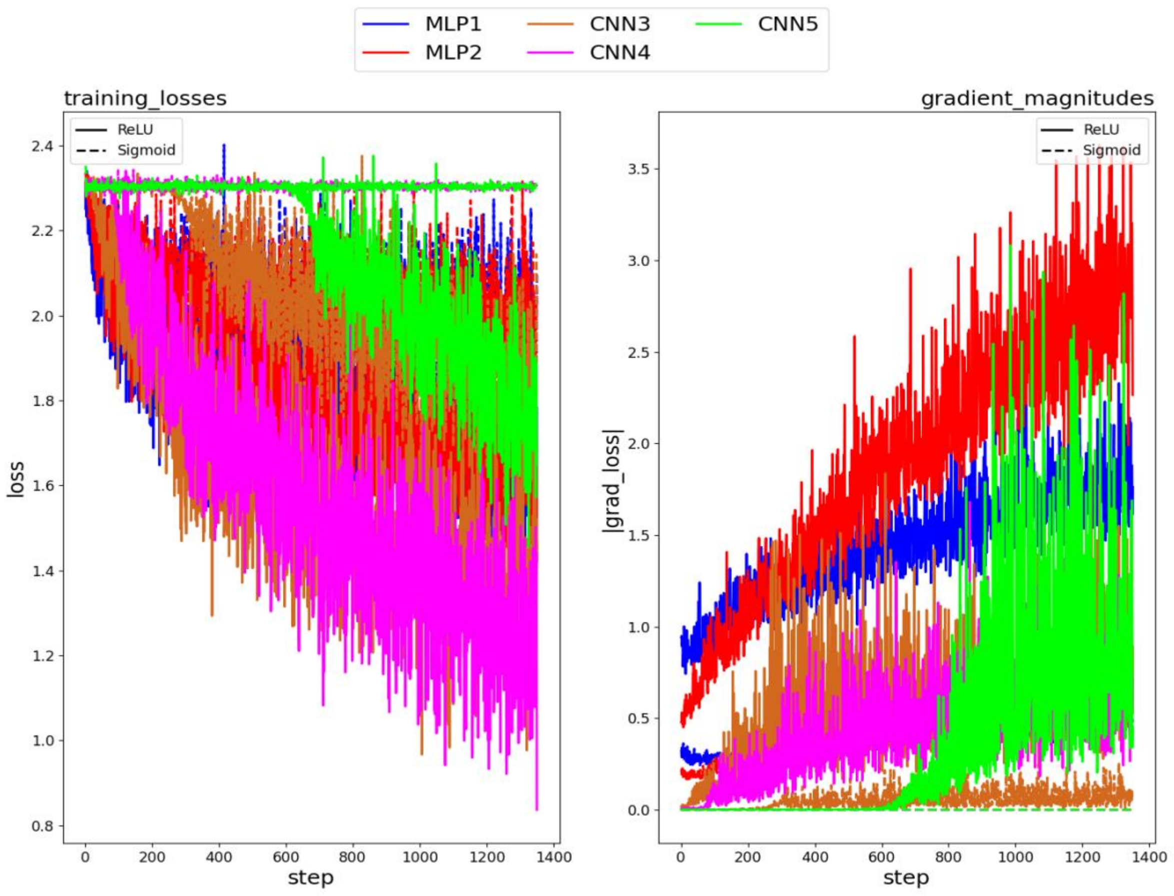


FIGURE 11. TRAINING LOSS AND GRADIENT MAGNITUDES VS. STEP PLOTS OF THE 5 ANN MODELS WITH 2 DIFFERENT ACTIVATION FUNCTIONS.

2. Discussions

1. In the plot, performance of the ANN models with different activation functions are observed. It can be seen that gradient magnitude is zero (or extremely small) for the model CNN5 with Sigmoid function as activation function. This phenomenon was mentioned earlier in part 3.2.5, as the vanishing gradient problem.

   Compared to the sigmoid activation function, the ReLU is less sensitive to the vanishing gradient problem. This is because ReLU provides results in the range of [0,1] and the derivative is either 0 or 1, depending on the input (0 for negative inputs, 1 for positive inputs). Derivation does not change as input increases; slope is always positive for positive inputs. However, that is not the case for the Sigmoid function. The rate of change of the sigmoid function decreases after some point as the input increases. The gradient signal can more easily propagate through ReLU units, allowing the network to learn more efficiently and quickly. Furthermore, the ReLU function is computationally efficient and can be easiliy implemented in hardware.

The derivative of the sigmoid function is always less then 1 and saturates as the input becomes extremely large or small. This makes it more challenging to update the weights effectively as the gradient value can become very small in the early layers of the network. Additionally, the sigmoid function is computationally expensive, involving the computation of exponential functions.

In the Artificial Neural Networks, gradient may become smaller and smaller as the depth increases, a phenomenon which leads to the vanishing gradient problem. The vanishing gradient problem occurs due to the fact that the gradient signal propagates backward through many layers, resulting a smaller and smaller gradient value. Hence, gradient will be closer to zero, as the depth increases.

2. As mentioned earlier, vanishing gradient problem which can be observed for the model CNN5 with Sigmoid function as activation function can be observed in the second plot in *Figure 11*. Depth of the models can be compared as follows:

$$Depth_{CNN5} > Depth_{CNN4} > Depth_{CNN3} > Depth_{MLP2} > Depth_{MLP1}$$

   This order also applies for the complexity since the complexity depends on the depth, number of parameters and layers. Gradient signal travels backwards all through the layers in the architecture, decreasing in magnitude at each step. This results in small gradient magnitude as the training continues.

3. An Artificial Neural Network (ANN) may take longer to get an adequate outcome or may not reach one at all if the inputs are in the range [0,255] and not scaled to the range [-1.0, 1.0] before training. This is due to the possibility that the wide range and distribution of the input characteristics may have an impact on how the weights and biases behave during training.

   When the input characteristics have varying ranges or scales, it's possible for certain characteristics to dominate over others, which leads the network to learn patterns that are biased towards those dominant features. Additionally, if the input features have high values, this might overwhelm the activation function and prevent the network from updating the weights efficiently.

5. Experimenting Learning Rate

1. Experimental Work

For the fifth part, I have chosen CNN4 for it has the best test accuracy among all the models. Firstly, CNN4 (with ReLU as activation function between layers) is trained three times for different values of learning rates; 0.1, 0.01, 0.001. Trainings are 20 epochs long. Other parameters are kept as before. After observing the training losses and validation accuracies for each training, the results are plotted, as it can be seen in *Figure 12*. It can be deducted that CNN4 performs best when the learning rate parameter is set to 0.1.

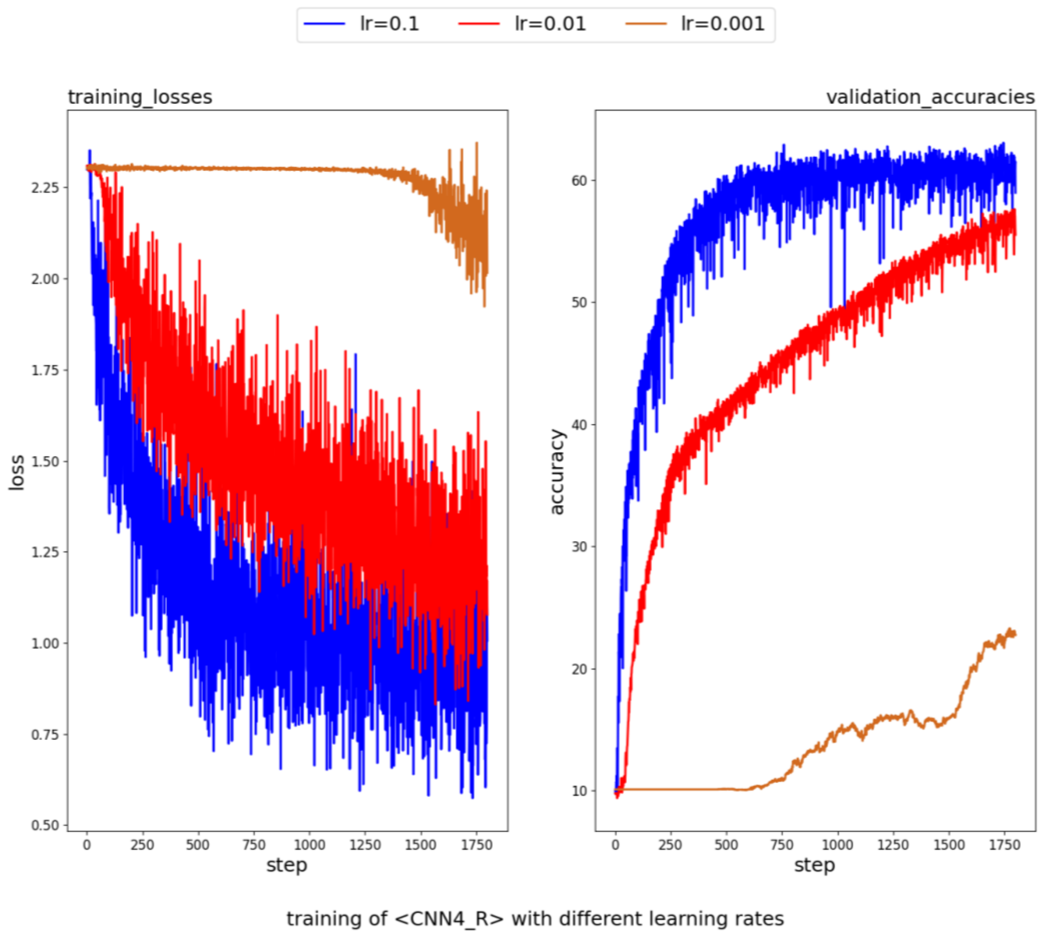
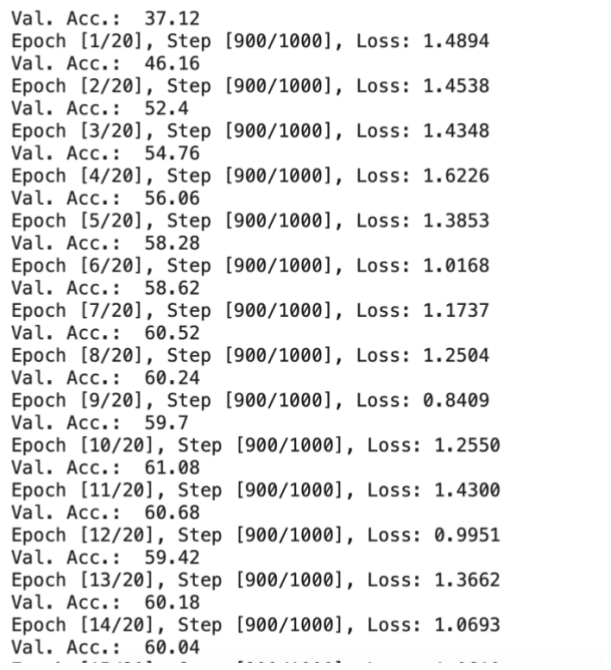


FIGURE 12. EXPERIMENTING ON THE LEARNING RATE, USING CNN4 ARCHITECTURE FOR PART 5.

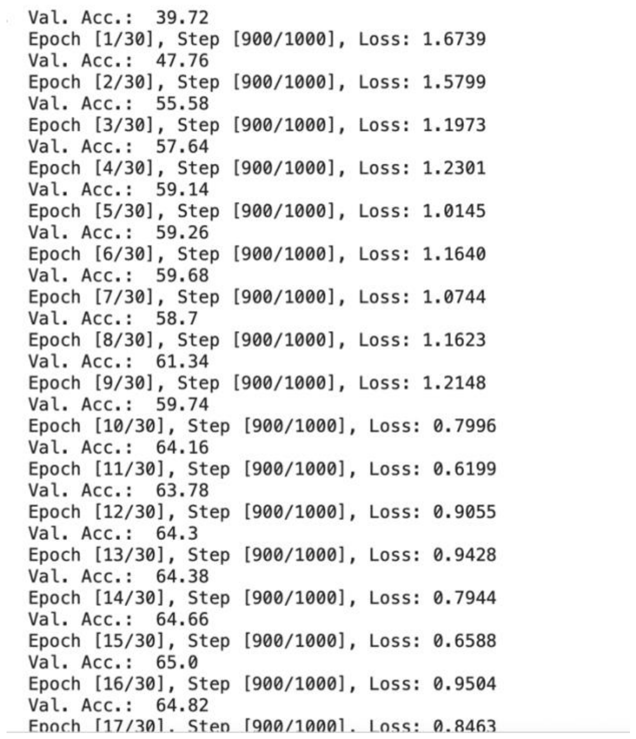
Inspecting the validation accuracy after each epoch during the training, it is observed that the validation accuracy and training loss converges to some value which can be observed in Figure 13. Validation accuracy gets stuck around 60 percent, although the training continues. This is not desired because it means that the model is not able to learn (or update the weights efficiently in a way that the classification capabilities increase) after a certain step in the training.

To increase the performance of our model, the step at which the validation accuracy improves no further, is recorded and the learning rate is changed at that significant epoch. After the 10th epoch, learning rate is changed to 0.01, number of epochs are increased to 30 and the results in Figure 14 are observed. This change boosted the validation accuracy to 64 percent. Hence, there is an improvement to some point on the classification ability of the model.



```
Val. Acc.:  37.12
Epoch [1/20], Step [900/1000], Loss: 1.4894
Val. Acc.:  46.16
Epoch [2/20], Step [900/1000], Loss: 1.4538
Val. Acc.:  52.4
Epoch [3/20], Step [900/1000], Loss: 1.4348
Val. Acc.:  54.76
Epoch [4/20], Step [900/1000], Loss: 1.6226
Val. Acc.:  56.06
Epoch [5/20], Step [900/1000], Loss: 1.3853
Val. Acc.:  58.28
Epoch [6/20], Step [900/1000], Loss: 1.0168
Val. Acc.:  58.62
Epoch [7/20], Step [900/1000], Loss: 1.1737
Val. Acc.:  60.52
Epoch [8/20], Step [900/1000], Loss: 1.2504
Val. Acc.:  60.24
Epoch [9/20], Step [900/1000], Loss: 0.8409
Val. Acc.:  59.7
Epoch [10/20], Step [900/1000], Loss: 1.2550
Val. Acc.:  61.08
Epoch [11/20], Step [900/1000], Loss: 1.4300
Val. Acc.:  60.68
Epoch [12/20], Step [900/1000], Loss: 0.9951
Val. Acc.:  59.42
Epoch [13/20], Step [900/1000], Loss: 1.3662
Val. Acc.:  60.18
Epoch [14/20], Step [900/1000], Loss: 1.0693
Val. Acc.:  60.04
```

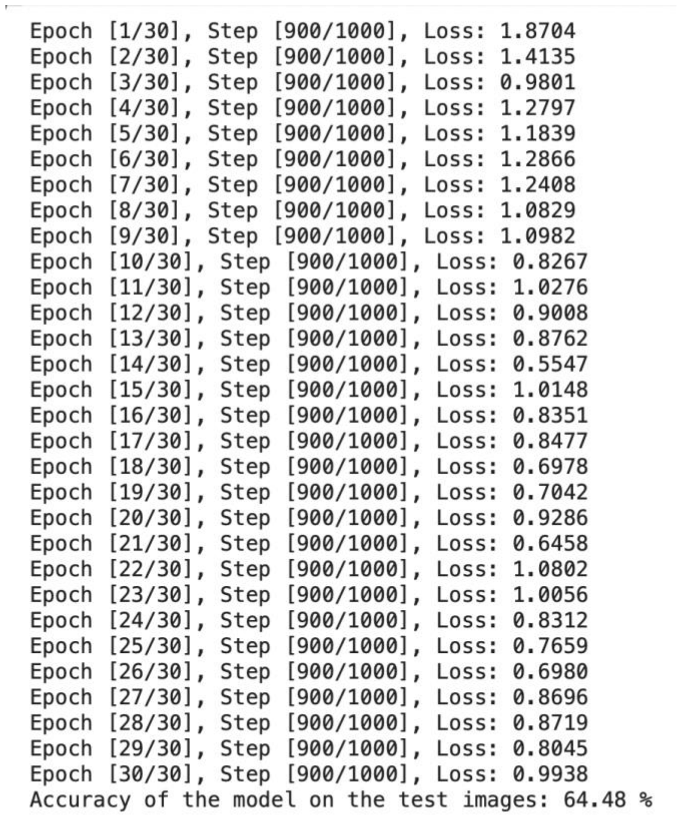
FIGURE 13. MONITORING OF THE TRAINING RESULTS OF CNN5 WHEN THE LEARNING RATE IS 0.1.



```
Val. Acc.:  39.72
Epoch [1/30], Step [900/1000], Loss: 1.6739
Val. Acc.:  47.76
Epoch [2/30], Step [900/1000], Loss: 1.5799
Val. Acc.:  55.58
Epoch [3/30], Step [900/1000], Loss: 1.1973
Val. Acc.:  57.64
Epoch [4/30], Step [900/1000], Loss: 1.2301
Val. Acc.:  59.14
Epoch [5/30], Step [900/1000], Loss: 1.0145
Val. Acc.:  59.26
Epoch [6/30], Step [900/1000], Loss: 1.1640
Val. Acc.:  59.68
Epoch [7/30], Step [900/1000], Loss: 1.0744
Val. Acc.:  58.7
Epoch [8/30], Step [900/1000], Loss: 1.1623
Val. Acc.:  61.34
Epoch [9/30], Step [900/1000], Loss: 1.2148
Val. Acc.:  59.74
Epoch [10/30], Step [900/1000], Loss: 0.7996
Val. Acc.:  64.16
Epoch [11/30], Step [900/1000], Loss: 0.6199
Val. Acc.:  63.78
Epoch [12/30], Step [900/1000], Loss: 0.9055
Val. Acc.:  64.3
Epoch [13/30], Step [900/1000], Loss: 0.9428
Val. Acc.:  64.38
Epoch [14/30], Step [900/1000], Loss: 0.7944
Val. Acc.:  64.66
Epoch [15/30], Step [900/1000], Loss: 0.6588
Val. Acc.:  65.0
Epoch [16/30], Step [900/1000], Loss: 0.9504
Val. Acc.:  64.82
Epoch [17/30], Step [900/1000], Loss: 0.8463
```

FIGURE 14. MONITORING OF THE TRAINING RESULTS OF CNN5 WHEN THE LEARNING RATE IS CHANGED TO 0.01 AFTER 10TH EPOCH.

Moreover, the step at which the validation accuracy converges to a certain percentage is recorded and the learning rate is changed to 0.001, after that step of the training. After the 15th epoch learning rate is changed to 0.001, number of epochs are increased to 30 and the results in Figure 15 are observed. Changing the learning rate one more time, had little effect on the validation accuracy. Using the model CNN4 and changing its learning rate parameter at certain epochs, increased the performance of the model to some point. Test accuracy of the final training is recorded as 64.48 percent.



```
Epoch [1/30], Step [900/1000], Loss: 1.8704
Epoch [2/30], Step [900/1000], Loss: 1.4135
Epoch [3/30], Step [900/1000], Loss: 0.9801
Epoch [4/30], Step [900/1000], Loss: 1.2797
Epoch [5/30], Step [900/1000], Loss: 1.1839
Epoch [6/30], Step [900/1000], Loss: 1.2866
Epoch [7/30], Step [900/1000], Loss: 1.2408
Epoch [8/30], Step [900/1000], Loss: 1.0829
Epoch [9/30], Step [900/1000], Loss: 1.0982
Epoch [10/30], Step [900/1000], Loss: 0.8267
Epoch [11/30], Step [900/1000], Loss: 1.0276
Epoch [12/30], Step [900/1000], Loss: 0.9008
Epoch [13/30], Step [900/1000], Loss: 0.8762
Epoch [14/30], Step [900/1000], Loss: 0.5547
Epoch [15/30], Step [900/1000], Loss: 1.0148
Epoch [16/30], Step [900/1000], Loss: 0.8351
Epoch [17/30], Step [900/1000], Loss: 0.8477
Epoch [18/30], Step [900/1000], Loss: 0.6978
Epoch [19/30], Step [900/1000], Loss: 0.7042
Epoch [20/30], Step [900/1000], Loss: 0.9286
Epoch [21/30], Step [900/1000], Loss: 0.6458
Epoch [22/30], Step [900/1000], Loss: 1.0802
Epoch [23/30], Step [900/1000], Loss: 1.0056
Epoch [24/30], Step [900/1000], Loss: 0.8312
Epoch [25/30], Step [900/1000], Loss: 0.7659
Epoch [26/30], Step [900/1000], Loss: 0.6980
Epoch [27/30], Step [900/1000], Loss: 0.8696
Epoch [28/30], Step [900/1000], Loss: 0.8719
Epoch [29/30], Step [900/1000], Loss: 0.8045
Epoch [30/30], Step [900/1000], Loss: 0.9938
Accuracy of the model on the test images: 64.48 %
```

FIGURE 15. CHANGING THE LEARNING RATE TO 0.001 AFTER THE 15TH EPOCH FOR THE CNN4 MODEL DURING THE TRAINING.

Examining the plot in Figure 16, it can be observed that there is significant increase (a jump) in validation accuracy around step = 1000. This is due to the change in learning rate. Changing the learning rate again at 15th epoch had little impact on the performance of the ANN model as it can be seen from Figure 17. However, the curve is observed to be more stable after the 15th epoch.

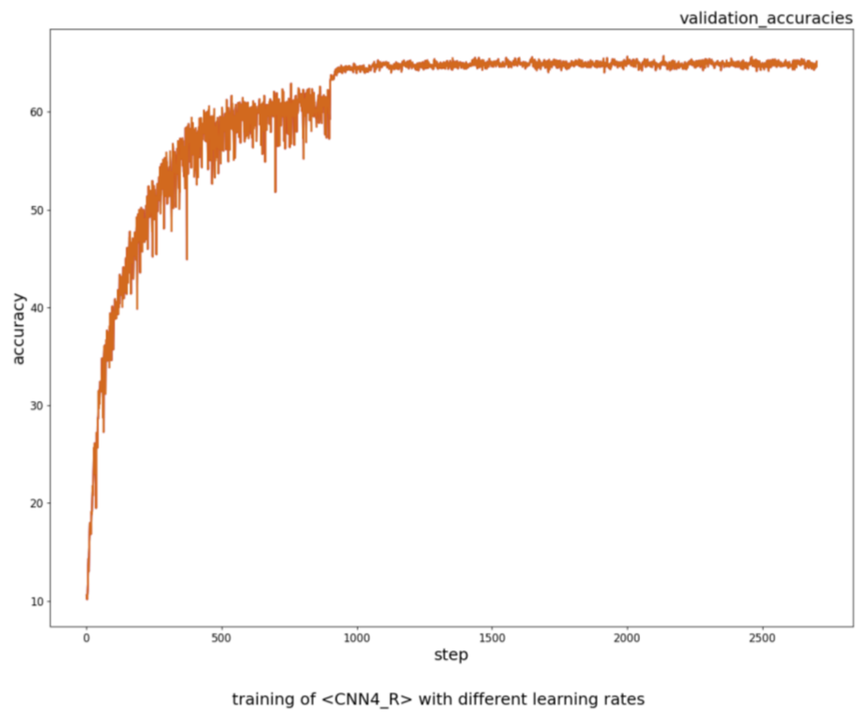


FIGURE 16. VALIDATION ACCURACY CURVE AS THE LEARNING RATE IS CHANGED TO 0.01 AT 10TH EPOCH.

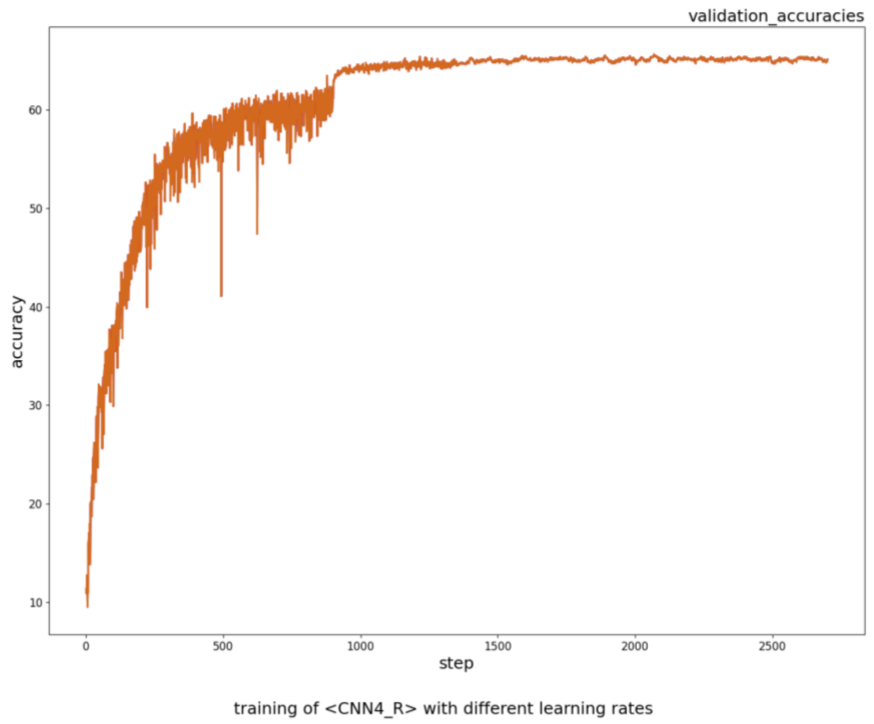


FIGURE 17. VALIDATION ACCURACY CURVE WHEN LEARNING RATE IS CHANGED CONCURRENTLY TO 0.01 AND 0.001 AT 10TH AND 15TH EPOCHS, RESPECTIVELY.

2. Discussions

1. The neural network will converge extremely slowly if the learning rate is too low since each iteration ends up resulting in a modest change for the weights. This may cause the network to converge slowly or cause it to become stuck in a non-ideal state.

   The network may converge rapidly, but it may not converge to a satisfactory solution if the learning rate is too high, on the other hand. This is because the weights may fluctuate or even deviate from the ideal values, making the network unstable and incapable of learning.

   Thus, choosing the proper learning rate is essential for obtaining fast and effective convergence.

2. If the learning rate is too low, the network may converge to a local minimum without getting a general look on the overall situation, without knowing the whereabouts of the global minimum. This may lead the network to a non-ideal point. On the other hand, if the learning rate is too high, the network may miss some local minima. This may lead to non-ideal change in weights. Consequently, the network provides inadequate performance since it is not trained efficiently.

   Utilizing adaptive learning rate techniques like Adam, Adagrad, or RMSprop, which modify learning rate based on gradient magnitude and history of gradient updates, is a frequent strategy in practice. By dynamically adjusting the learning rate and enhancing the network's stability and convergence speed, these techniques can aid in accelerating convergence to a better point.

   Instead of using an optimizer that has the adaptive features as Adam, SGD is used for the part 5, and the learning rate is changed manually at different epochs during the training. Changing the learning rate manually is less efficient comparing to adaptive learning rate techniques. However, observing how the performance of an ANN Classifier is affected by dynamically changed learning rate parameter, was highly informative.

3. Scheduled learning rate method worked and improved the performance of the ANN Classifier to some point. When number of epochs were 20 and learning set to 0.1, the validation accuracy was around 60 percent. When number of epoch is increased to 30 and learning rate changed to 0.01 at 10th epoch and 0.001 at 15th epoch, validation accuracy increased by nearly 5 percent. Finally, observed test accuracy of the ANN model, CNN4, with scheduled learning rate, is 64.48 percent. It can be deducted that changing the learning rate parameter during the training process can affect the performance in a good way. Moreover, changing the learning rate dynamically (adaptive learning rate) will provide the most efficient results. However, more computation power may be needed, and the training may take longer to finish

with such an optimizer.

4. In the third part, we have utilized the Adam optimizer for the ANN classifiers. Training the models 10 times (20 epochs for each), different test accuracies are observed. For CNN4, best test accuracy among ten trainings was equal to 63.42 percent.

   In the fifth part, SGD optimizer is used with learning rate of 0.1 and 60 percent of test accuracy is observed for CNN4. Changing the learning rates at certain epochs, had a positive impact on the performance of the classifier, and the test accuracy was increased to 64.48 percent. Meaning that, training done with manually changed learning rate, provided slightly better results than the training done with Adam optimizer.

Remark I: I used Google Colab, writing and running the codes. Codes are written cell by cell since it is easier to run specific lines of codes. Therefore, some libraries are imported multiple times for the same question, to only run that specific cell.

Remark II: Plotting functions are not added in the Appendices. However, it must be added to the code, in order to use those functions.

APPENDIX I

```
### Question 2 ###

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.lines import Line2D
import os
import torch as nn
from torchvision.utils import make_grid

# input shape: [batch size, input_channels, input_height, input_width]
input=np.load("samples_1.npy")
# input shape: [output_channels, input_channels, filter_height, filter width]
kernel=np.load("kernel.npy")

# Observing the dimensions of the input and the kernel
print("kernel shape: ", kernel.shape)
print("input shape: ", input.shape)


def conv2d(input_data, kernel):
    # Get the dimensions of the input data and kernel
    input_shape = input_data.shape
    kernel_shape = kernel.shape

    # Compute the output shape based on the input and kernel shapes
    output_shape = (
    input_shape[0], kernel_shape[0], input_shape[2] - kernel_shape[2] + 1,
input_shape[3] - kernel_shape[3] + 1)

    # Create an empty array to hold the output data
    output_data = np.zeros(output_shape)

    # Perform 2D convolution
    for i in range(output_shape[0]):
        for j in range(kernel_shape[0]):
            for k in range(output_shape[2]):
                for l in range(output_shape[3]):
                    output_data[i][j][k][l] = np.sum(
                        input_data[i][..., k:k + kernel_shape[2], l:l +
kernel_shape[3]] * kernel[j])

    return output_data


out = conv2d(input, kernel)
```

```
# Normalize the output
out = (out - np.min(out)) / (np.max(out) - np.min(out))

part2Plots(out=out, save_dir="results", filename="1_normalized")

### End of Question 2 ###
```

APPENDIX II

```
### Question 3 ###


# Import PyTorch library and optimization module
import torch
import torch.optim as optim

# Import torchvision library and numpy
import torchvision
import numpy as np
import torchvision.transforms as transforms

# Import train_test_split from scikit-learn
from sklearn.model_selection import train_test_split

# Check if GPU is available, else use CPU
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# CIFAR-10 images have a size of 32x32
# Define a series of image transformations to be applied to the CIFAR-10 dataset
transform = transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
        torchvision.transforms.Grayscale()
])

# Load the CIFAR-10 dataset into a PyTorch data loader, applying the defined
transformations to it
train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True,
transform=transform)

# Split the training data into a training set and a validation set, with a split ratio
of 90:10
train_set, val_data = train_test_split(train_data, test_size=0.1, random_state=42)

# Create a PyTorch data loader for the training set
train_generator = torch.utils.data.DataLoader(train_set, batch_size=50, shuffle=True)
```

```python
# Load the CIFAR-10 test dataset into a PyTorch data loader, applying the same
transformations as for the training data
test_data = torchvision.datasets.CIFAR10('./data', train=False, transform=transform)

# Create a PyTorch data loader for the test set
test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)

# Create a PyTorch data loader for the validation set
val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)


# Define the MLP models
class MLP1(torch.nn.Module):
    def __init__(self):
        super(MLP1, self).__init__()
        # Defining the Layers
        self.fc1 = torch.nn.Linear(1024, 32,bias = True)
        self.fc2 = torch.nn.Linear(32, 10,bias = True)

    def forward(self, x):
        # Connecting the layers
        x = x.view(-1, 1024)  # Flatten input tensor
        #print('x shape: ', x.shape)
        # x is now [50,1024] since batch size is 50
        # and we linearize the input 32*32 (input size) to 1024*1
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        return x

class MLP2(torch.nn.Module):
    def __init__(self):
        super(MLP2, self).__init__()
        self.fc1 = torch.nn.Linear(1024, 32,bias = True)
        self.fc2 = torch.nn.Linear(32, 64, bias = False)
        self.prediction_layer = torch.nn.Linear(64, 10,bias = False)

    def forward(self, x):

        x = x.view(-1, 1024)  # Flatten input tensor
        #print('x shape: ', x.shape)
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        x = self.prediction_layer(x)
        return x


# Define the CNN models
# To track the dimensions of the outputs of the layers,
# print(x.shape) is used between the lines
class CNN3(torch.nn.Module):
    def __init__(self):
        super(CNN3, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 5x5x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 7x7x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(7,7), stride = 1, padding = 'valid')
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*3*3, out_features=10)

        # (height,width) - kernel_size + 1 = output dimensions !!!

    def forward(self, x):
        # torch.Size([50, 1, 32, 32]) , 50 32x32 images in one batch
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30]) , after CONV.3x3x16
        x = self.relu1(x)
        # torch.Size([50, 16, 30, 30]), RelU does NOT effect the dimensions
        x = self.conv2(x)
        # torch.Size([50, 8, 26, 26]) , after CONV.5x5x8
        x = self.relu2(x)
        x = self.maxpool(x)
        # torch.Size([50, 8, 13, 13]) , after MaxPool2d (height and width are two
times smaller)
        x = self.conv3(x)
        # torch.Size([50, 16, 7, 7]) , after CONV.7x7x16
        x = self.maxpool(x)
        # torch.Size([50, 16, 3, 3]) , after MaxPool2d
        x = x.view(50 , 16*3*3)
        # torch.Size([50, 144]) , after x.view
        x = self.prediction_layer(x)

        return x

class CNN4(torch.nn.Module):
    def __init__(self):
        super(CNN4, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # Conv. 5x5x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu3 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 5x5x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu4 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding =
0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*4*4, out_features=10)

    def forward(self, x):
        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30])
        x = self.relu1(x)
        x = self.conv2(x)
        # torch.Size([50, 8, 28, 28])
        x = self.relu2(x)
        x = self.conv3(x)
        # torch.Size([50, 16, 24, 24])
        x = self.relu3(x)
        x = self.maxpool1(x)
        # torch.Size([50, 16, 12, 12])
        x = self.conv4(x)
        x = self.relu4(x)
        # torch.Size([50, 16, 8, 8])
        x = self.maxpool2(x)
        # torch.Size([50, 16, 4, 4])
        x = x.view(50 ,16*4*4)
        # torch.Size([50, 256])
        x = self.prediction_layer(x)
        return x
```

```
class CNN5(torch.nn.Module):
    def __init__(self):
        super(CNN5, self).__init__()
        # Conv. 3x3x8 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=8, kernel_size= (3,3)
, stride = 1, padding='valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 3x3x16 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding='valid')
        # ReLU layer
        self.relu3 = torch.nn.ReLU()
        # Conv. 3x3x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # ReLU layer
        self.relu4 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 3x3x16 Layer
        self.conv5 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # ReLU layer
        self.relu5 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv6 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding='valid')
        # ReLU layer
        self.relu6 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 8*4*4, out_features=10)


    def forward(self, x):

        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # Size = [50, 8, 30, 30] (guessed)
        x = self.relu1(x)
        x = self.conv2(x)
        # Size = [50, 16, 28, 28] (guessed)
        x = self.relu2(x)
        x = self.conv3(x)
        # Size = [50, 8, 26, 26] (guessed)
        x = self.relu3(x)
        x = self.conv4(x)
        # Size = [50, 16, 24, 24] (guessed)
        x = self.relu4(x)
        x = self.maxpool1(x)
        # Size = [50, 16, 12, 12] (guessed)
        # MaxPool does NOT efect number of output channels
        x = self.conv5(x)
        # Size = [50, 16, 10, 10] (guessed)
        x = self.relu5(x)
        x = self.conv6(x)
        # Size = [50, 8, 8, 8] (guessed)
        x = self.relu6(x)
        x = self.maxpool2(x)
        # Size = [50, 8, 4, 4] (guessed)
        # torch.Size([50, 8, 4, 4]) (the guess was correct)
        x = x.view(50 ,8*4*4)

        x = self.prediction_layer(x)


        return x


# Model name (Change the model and model_name)
model_name = 'MLP1'
# Number of tests
num_test = 10

# Batch size is 50
batch_size = 50
# Number of epochs is 15
num_epochs = 15

# Each model is trained 10 times
for x in range(num_test):

  # Create model, loss function, and optimizer
  model = MLP1()
  model.to(device)
  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters())

  # Initialize the lists
  train_loss_history = []
  train_acc_history = []
  val_acc_history = []
```

```
for epoch in range(num_epochs):

    for i, (images, labels) in enumerate(train_generator):
        # Convert data to PyTorch tensors
        images = images.numpy()
        labels = labels.numpy()
        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass and compute loss
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        # Record training accuracy every 10 steps
        if (i+1) % 10 == 0:

            # Record training loss
            train_loss_history.append(loss.item())

            with torch.no_grad():
                total = 0
                correct = 0

                #outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                train_accuracy = 100 * correct / total
                train_acc_history.append(train_accuracy)

                # Record validation accuracy every 10 steps
                total = 0
                correct = 0
                for images, labels in val_generator:
                    images = images.to(device)
                    labels = labels.to(device)
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
                val_accuracy = 100 * correct / total
```

```
                  val_acc_history.append(val_accuracy)

          # Print training loss every 100 steps
          if (i+1) % 100 == 0:
              print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                     .format(epoch+1, num_epochs, i+1, len(train_data)//batch_size,
loss.item()))


  print(len(train_loss_history))
  print(len(train_acc_history))
  print(len(val_acc_history))


  # Test the model
  model.eval()
  with torch.no_grad():
      correct = 0
      total = 0
      for images, labels in test_generator:
          # Convert data to PyTorch tensors
          images = images.numpy()
          labels = labels.numpy()
          images = torch.from_numpy(images)
          labels = torch.from_numpy(labels)
          images = images.to(device)
          labels = labels.to(device)

          # Compute predictions
          outputs = model(images)
          _, predicted = torch.max(outputs.data, 1)
          total += labels.size(0)
          correct += (predicted == labels).sum().item()
      test_acc = 100 * correct / total
      print('Accuracy of the model on the test images: {} %'.format(test_acc))

  import pickle

  # create the dictionary object with the required key-value pairs
  result_dict = {
      'name': model_name,
      'loss curve': train_loss_history,
      'train acc curve': train_acc_history,
      'val acc curve': val_acc_history,
      'test acc': test_acc,
      'weights': model.conv1.weight.data
  }

  # save the dictionary object to a file
  filename = 'part3_'+str(model_name)+'_'+str(x+1)+'.pkl'
  with open(filename, 'wb') as file:
      pickle.dump(result_dict, file)

  # load the dictionary object from the file
  with open(filename, 'rb') as file:
      loaded_dict = pickle.load(file)

  # print the loaded dictionary object
  print(loaded_dict)


import pickle

# After traning each model 10 times,
# It is time to take the average of the
# loss curve, train accuracy, and validation accuracy
# This part is used 5 times for each model

# Model Name
model_name = 'CNN5'

file_name1 = 'part3_'+str(model_name)+'_1.pkl'
file_name2 = 'part3_'+str(model_name)+'_2.pkl'
file_name3 = 'part3_'+str(model_name)+'_3.pkl'
file_name4 = 'part3_'+str(model_name)+'_4.pkl'
file_name5 = 'part3_'+str(model_name)+'_5.pkl'
file_name6 = 'part3_'+str(model_name)+'_6.pkl'
file_name7 = 'part3_'+str(model_name)+'_7.pkl'
file_name8 = 'part3_'+str(model_name)+'_8.pkl'
file_name9 = 'part3_'+str(model_name)+'_9.pkl'
file_name10 = 'part3_'+str(model_name)+'_10.pkl'

# load the dictionary object from the file
with open(file_name1, 'rb') as file:
  loaded_dict1 = pickle.load(file)

# load the dictionary object from the file
with open(file_name2, 'rb') as file:
  loaded_dict2 = pickle.load(file)

# load the dictionary object from the file
with open(file_name3, 'rb') as file:
  loaded_dict3 = pickle.load(file)

# load the dictionary object from the file
with open(file_name4, 'rb') as file:
  loaded_dict4 = pickle.load(file)

# load the dictionary object from the file
with open(file_name5, 'rb') as file:
  loaded_dict5 = pickle.load(file)

# load the dictionary object from the file
with open(file_name6, 'rb') as file:
  loaded_dict6 = pickle.load(file)

# load the dictionary object from the file
with open(file_name7, 'rb') as file:
  loaded_dict7 = pickle.load(file)

# load the dictionary object from the file
with open(file_name8, 'rb') as file:
  loaded_dict8 = pickle.load(file)

# load the dictionary object from the file
with open(file_name9, 'rb') as file:
  loaded_dict9 = pickle.load(file)

# load the dictionary object from the file
with open(file_name10, 'rb') as file:
  loaded_dict10 = pickle.load(file)

dict_name = 'loss curve'
lists = [loaded_dict1[dict_name], loaded_dict2[dict_name], loaded_dict3[dict_name],
         loaded_dict4[dict_name], loaded_dict5[dict_name], loaded_dict6[dict_name],
         loaded_dict7[dict_name], loaded_dict8[dict_name], loaded_dict9[dict_name],
loaded_dict10[dict_name]]

# Initialize an empty list to store the averages
loss_curve_average = []

# Use a for loop to iterate over each index in the lists
for i in range(len(loaded_dict1[dict_name])):
    # Initialize a variable to store the sum of the values at that index
    total = 0
    # Use another for loop to iterate over each list in the list of lists
    for lst in lists:
        # Add the value at the current index to the total
        total += lst[i]
    # Calculate the average by dividing the total by the number of lists
    avg = total / len(lists)
    # Append the average to the averages list
    loss_curve_average.append(avg)

dict_name = 'train acc curve'
lists = [loaded_dict1[dict_name], loaded_dict2[dict_name], loaded_dict3[dict_name],
         loaded_dict4[dict_name], loaded_dict5[dict_name], loaded_dict6[dict_name],
         loaded_dict7[dict_name], loaded_dict8[dict_name], loaded_dict9[dict_name],
loaded_dict10[dict_name]]
```

```
# Initialize an empty list to store the averages
train_acc_curve_average = []

# Use a for loop to iterate over each index in the lists
for i in range(len(loaded_dict1[dict_name])):
    # Initialize a variable to store the sum of the values at that index
    total = 0
    # Use another for loop to iterate over each list in the list of lists
    for lst in lists:
        # Add the value at the current index to the total
        total += lst[i]
    # Calculate the average by dividing the total by the number of lists
    avg = total / len(lists)
    # Append the average to the averages list
    train_acc_curve_average.append(avg)



dict_name = 'val acc curve'
lists = [loaded_dict1[dict_name], loaded_dict2[dict_name], loaded_dict3[dict_name],
         loaded_dict4[dict_name], loaded_dict5[dict_name], loaded_dict6[dict_name],
         loaded_dict7[dict_name], loaded_dict8[dict_name], loaded_dict9[dict_name],
loaded_dict10[dict_name]]

# Initialize an empty list to store the averages
val_acc_curve_average = []

# Use a for loop to iterate over each index in the lists
for i in range(len(loaded_dict1[dict_name])):
    # Initialize a variable to store the sum of the values at that index
    total = 0
    # Use another for loop to iterate over each list in the list of lists
    for lst in lists:
        # Add the value at the current index to the total
        total += lst[i]
    # Calculate the average by dividing the total by the number of lists
    avg = total / len(lists)
    # Append the average to the averages list
    val_acc_curve_average.append(avg)

# Finding out which of the runs provides best performance
print(loaded_dict1['test acc'])
print(loaded_dict2['test acc'])
print(loaded_dict3['test acc'])
print(loaded_dict4['test acc'])
print(loaded_dict5['test acc'])
print(loaded_dict6['test acc'])
print(loaded_dict7['test acc'])
print(loaded_dict8['test acc'])
print(loaded_dict9['test acc'])
print(loaded_dict10['test acc'])

# For MLP1, (40.18) second run
# For MLP2, (39.78) first run
# For CNN3, (59.42) tenth run
# For CNN4, (63.42) tenth run
# For CNN5, (58.84) sixth run, gives best test accuracy (performance)


print(loaded_dict6['name'])
best_performance = loaded_dict6['test acc']
weights_of_first_layer = loaded_dict6['weights']


# Creating one final pickle folder for each model

# create the dictionary object with the required key-value pairs
result = {
    'name': model_name,
    'loss_curve': loss_curve_average,
    'train_acc_curve': train_acc_curve_average,
    'val_acc_curve': val_acc_curve_average,
    'test_acc': best_performance,
    'weights': weights_of_first_layer
}

# save the dictionary object to a file
filename = str(model_name)+'.pkl'
with open(filename, 'wb') as file:
    pickle.dump(result, file)


import pickle

# load the dictionary object from the file
with open('MLP1.pkl', 'rb') as file:
  MLP1 = pickle.load(file)
with open('MLP2.pkl', 'rb') as file:
  MLP2 = pickle.load(file)
with open('CNN3.pkl', 'rb') as file:
  CNN3 = pickle.load(file)
with open('CNN4.pkl', 'rb') as file:
  CNN4 = pickle.load(file)
with open('CNN5.pkl', 'rb') as file:
  CNN5 = pickle.load(file)

dict_models = [MLP1,MLP2,CNN3,CNN4,CNN5]

print('### Test Accuracies ###')
print('MLP1: ',MLP1['test_acc'])
print('MLP2: ',MLP2['test_acc'])
print('CNN3: ',CNN3['test_acc'])
print('CNN4: ',CNN4['test_acc'])
print('CNN5: ',CNN5['test_acc'])

part3Plots(dict_models, save_dir=r'data', filename='part3Plots')

weight = MLP2['weights'].cpu().numpy()
visualizeWeights(weight, save_dir='data', filename='weight_plot_MLP2')


### End of Question 3 ###
```

APPENDIX III

```
### Question 4 ###

# Import PyTorch library and optimization module
import torch
import torch.optim as optim

# Import torchvision library and numpy
import torchvision
import numpy as np
import torchvision.transforms as transforms

# Import train_test_split from scikit-learn
from sklearn.model_selection import train_test_split

# Check if GPU is available, else use CPU
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# CIFAR-10 images have a size of 32x32


# Define a series of image transformations to be applied to the CIFAR-10 dataset
transform = transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
        torchvision.transforms.Grayscale()
])

# Load the CIFAR-10 dataset into a PyTorch data loader, applying the defined
transformations to it
```

```
train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True,
transform=transform)

# Split the training data into a training set and a validation set, with a split ratio
of 90:10
train_set, val_data = train_test_split(train_data, test_size=0.1, random_state=42)

# Create a PyTorch data loader for the training set
train_generator = torch.utils.data.DataLoader(train_set, batch_size=50, shuffle=True)

# Load the CIFAR-10 test dataset into a PyTorch data loader, applying the same
transformations as for the training data
test_data = torchvision.datasets.CIFAR10('./data', train=False, transform=transform)

# Create a PyTorch data loader for the test set
test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)

# Create a PyTorch data loader for the validation set
val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)


# Define the MLP models
class MLP1_R(torch.nn.Module):
    def __init__(self):
        super(MLP1_R, self).__init__()
        # Defining the Layers
        self.fc1 = torch.nn.Linear(1024, 32,bias = True)
        self.fc2 = torch.nn.Linear(32, 10,bias = True)

    def forward(self, x):
        # Connecting the layers
        x = x.view(-1, 1024)  # Flatten input tensor
        #print('x shape: ', x.shape)
        # x is now [50,1024] since batch size is 50
        # and we linearize the input 32*32 (input size) to 1024*1
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        return x

class MLP1_S(torch.nn.Module):
    def __init__(self):
        super(MLP1_S, self).__init__()
        # Defining the Layers
        self.fc1 = torch.nn.Linear(1024, 32,bias = True)
        self.fc2 = torch.nn.Linear(32, 10,bias = True)

    def forward(self, x):
        # Connecting the layers
        x = x.view(-1, 1024)  # Flatten input tensor
        #print('x shape: ', x.shape)
        # x is now [50,1024] since batch size is 50
        # and we linearize the input 32*32 (input size) to 1024*1
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x

class MLP2_R(torch.nn.Module):
    def __init__(self):
        super(MLP2_R, self).__init__()
        self.fc1 = torch.nn.Linear(1024, 32,bias = True)
        self.fc2 = torch.nn.Linear(32, 64, bias = False)
        self.prediction_layer = torch.nn.Linear(64, 10,bias = False)

    def forward(self, x):

        x = x.view(-1, 1024)  # Flatten input tensor
        #print('x shape: ', x.shape)
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        x = self.prediction_layer(x)
        return x

class MLP2_S(torch.nn.Module):
    def __init__(self):
        super(MLP2_S, self).__init__()
        self.fc1 = torch.nn.Linear(1024, 32,bias = True)
        self.fc2 = torch.nn.Linear(32, 64, bias = False)
        self.prediction_layer = torch.nn.Linear(64, 10,bias = False)

    def forward(self, x):

        x = x.view(-1, 1024)  # Flatten input tensor
        #print('x shape: ', x.shape)
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        x = self.prediction_layer(x)
        return x


# Define the CNN models
# To track the dimensions of the outputs of the layers,
# print(x.shape) is used between the lines
class CNN3_R(torch.nn.Module):
    def __init__(self):
        super(CNN3_R, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 5x5x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 7x7x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(7,7), stride = 1, padding = 'valid')
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*3*3, out_features=10)

        # (height,width) - kernel_size + 1 = output dimensions !!!

    def forward(self, x):
        # torch.Size([50, 1, 32, 32]) , 50 32x32 images in one batch
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30]) , after CONV.3x3x16
        x = self.relu1(x)
        # torch.Size([50, 16, 30, 30]), RelU does NOT effect the dimensions
        x = self.conv2(x)
        # torch.Size([50, 8, 26, 26]) , after CONV.5x5x8
        x = self.relu2(x)
        x = self.maxpool(x)
        # torch.Size([50, 8, 13, 13]) , after MaxPool2d (height and width are two
times smaller)
        x = self.conv3(x)
        # torch.Size([50, 16, 7, 7]) , after CONV.7x7x16
        x = self.maxpool(x)
        # torch.Size([50, 16, 3, 3]) , after MaxPool2d
        x = x.view(50 , 16*3*3)
        # torch.Size([50, 144]) , after x.view
        x = self.prediction_layer(x)

        return x

class CNN3_S(torch.nn.Module):
    def __init__(self):
        super(CNN3_S, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # Sigmoid layer
        self.sigmoid1 = torch.nn.Sigmoid()
        # Conv. 5x5x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # Sigmoid layer
        self.sigmoid2 = torch.nn.Sigmoid()
        # MaxPool 2x2
        self.maxpool = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 7x7x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(7,7), stride = 1, padding = 'valid')
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*3*3, out_features=10)

        # (height,width) - kernel_size + 1 = output dimensions !!!

    def forward(self, x):
        # torch.Size([50, 1, 32, 32]) , 50 32x32 images in one batch
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30]) , after CONV.3x3x16
        x = self.sigmoid1(x)
        # torch.Size([50, 16, 30, 30]), RelU does NOT effect the dimensions
        x = self.conv2(x)
        # torch.Size([50, 8, 26, 26]) , after CONV.5x5x8
        x = self.sigmoid2(x)
        x = self.maxpool(x)
        # torch.Size([50, 8, 13, 13]) , after MaxPool2d (height and width are two
times smaller)
        x = self.conv3(x)
        # torch.Size([50, 16, 7, 7]) , after CONV.7x7x16
        x = self.maxpool(x)
        # torch.Size([50, 16, 3, 3]) , after MaxPool2d
        x = x.view(50 , 16*3*3)
        # torch.Size([50, 144]) , after x.view
        x = self.prediction_layer(x)

        return x

class CNN4_R(torch.nn.Module):
    def __init__(self):
        super(CNN4_R, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # Conv. 5x5x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu3 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 5x5x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu4 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding =
0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*4*4, out_features=10)

    def forward(self, x):
        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30])
        x = self.relu1(x)
        x = self.conv2(x)
        # torch.Size([50, 8, 28, 28])
        x = self.relu2(x)
        x = self.conv3(x)
        # torch.Size([50, 16, 24, 24])
        x = self.relu3(x)
        x = self.maxpool1(x)
        # torch.Size([50, 16, 12, 12])
        x = self.conv4(x)
        x = self.relu4(x)
        # torch.Size([50, 16, 8, 8])
        x = self.maxpool2(x)
        # torch.Size([50, 16, 4, 4])
        x = x.view(50 ,16*4*4)
        # torch.Size([50, 256])
        x = self.prediction_layer(x)
        return x

class CNN4_S(torch.nn.Module):
    def __init__(self):
        super(CNN4_S, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # Sigmoid layer
        self.sigmoid1 = torch.nn.Sigmoid()
        # Conv. 3x3x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding = 'valid')
        # Sigmoid layer
        self.sigmoid2 = torch.nn.Sigmoid()
        # Conv. 5x5x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # Sigmoid layer
        self.sigmoid3 = torch.nn.Sigmoid()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 5x5x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(5,5), stride = 1, padding = 'valid')
        # Sigmoid layer
        self.sigmoid4 = torch.nn.Sigmoid()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding =
0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*4*4, out_features=10)

    def forward(self, x):
        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30])
        x = self.sigmoid1(x)
        x = self.conv2(x)
        # torch.Size([50, 8, 28, 28])
        x = self.sigmoid2(x)
        x = self.conv3(x)
        # torch.Size([50, 16, 24, 24])
        x = self.sigmoid3(x)
        x = self.maxpool1(x)
        # torch.Size([50, 16, 12, 12])
        x = self.conv4(x)
        x = self.sigmoid4(x)
        # torch.Size([50, 16, 8, 8])
        x = self.maxpool2(x)
        # torch.Size([50, 16, 4, 4])
        x = x.view(50 ,16*4*4)
        # torch.Size([50, 256])
        x = self.prediction_layer(x)
        return x


class CNN5_R(torch.nn.Module):
    def __init__(self):
        super(CNN5_R, self).__init__()
        # Conv. 3x3x8 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=8, kernel_size= (3,3)
, stride = 1, padding='valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 3x3x16 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding='valid')
        # ReLU layer
        self.relu3 = torch.nn.ReLU()
        # Conv. 3x3x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # ReLU layer
        self.relu4 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 3x3x16 Layer
        self.conv5 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # ReLU layer
        self.relu5 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv6 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding='valid')
        # ReLU layer
        self.relu6 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 8*4*4, out_features=10)


    def forward(self, x):

        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # Size = [50, 8, 30, 30] (guessed)
        x = self.relu1(x)
        x = self.conv2(x)
        # Size = [50, 16, 28, 28] (guessed)
        x = self.relu2(x)
        x = self.conv3(x)
        # Size = [50, 8, 26, 26] (guessed)
        x = self.relu3(x)
        x = self.conv4(x)
        # Size = [50, 16, 24, 24] (guessed)
        x = self.relu4(x)
        x = self.maxpool1(x)
        # Size = [50, 16, 12, 12] (guessed)
        # MaxPool does NOT efect number of output channels
        x = self.conv5(x)
        # Size = [50, 16, 10, 10] (guessed)
        x = self.relu5(x)
        x = self.conv6(x)
        # Size = [50, 8, 8, 8] (guessed)
        x = self.relu6(x)
        x = self.maxpool2(x)
        # Size = [50, 8, 4, 4] (guessed)
        # torch.Size([50, 8, 4, 4]) (the guess was correct)
        x = x.view(50 ,8*4*4)

        x = self.prediction_layer(x)


        return x


class CNN5_S(torch.nn.Module):
    def __init__(self):
        super(CNN5_S, self).__init__()
        # Conv. 3x3x8 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=8, kernel_size= (3,3)
, stride = 1, padding='valid')
        # Sigmoid layer
        self.sigmoid1 = torch.nn.Sigmoid()
        # Conv. 3x3x16 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # Sigmoid layer
        self.sigmoid2 = torch.nn.Sigmoid()
        # Conv. 3x3x8 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding='valid')
        # Sigmoid layer
        self.sigmoid3 = torch.nn.Sigmoid()
        # Conv. 3x3x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
        # Sigmoid layer
        self.sigmoid4 = torch.nn.Sigmoid()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 3x3x16 Layer
        self.conv5 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(3,3) , stride = 1, padding='valid')
```

```
        # Sigmoid layer
        self.sigmoid5 = torch.nn.Sigmoid()
        # Conv. 3x3x8 Layer
        self.conv6 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding='valid')
        # Sigmoid layer
        self.sigmoid6 = torch.nn.Sigmoid()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 8*4*4, out_features=10)


    def forward(self, x):

        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # Size = [50, 8, 30, 30] (guessed)
        x = self.sigmoid1(x)
        x = self.conv2(x)
        # Size = [50, 16, 28, 28] (guessed)
        x = self.sigmoid2(x)
        x = self.conv3(x)
        # Size = [50, 8, 26, 26] (guessed)
        x = self.sigmoid3(x)
        x = self.conv4(x)
        # Size = [50, 16, 24, 24] (guessed)
        x = self.sigmoid4(x)
        x = self.maxpool1(x)
        # Size = [50, 16, 12, 12] (guessed)
        # MaxPool does NOT efect number of output channels
        x = self.conv5(x)
        # Size = [50, 16, 10, 10] (guessed)
        x = self.sigmoid5(x)
        x = self.conv6(x)
        # Size = [50, 8, 8, 8] (guessed)
        x = self.sigmoid6(x)
        x = self.maxpool2(x)
        # Size = [50, 8, 4, 4] (guessed)
        # torch.Size([50, 8, 4, 4]) (the guess was correct)
        x = x.view(50 ,8*4*4)

        x = self.prediction_layer(x)


        return x


# Change the model and the 2 models to be trained (for RelU and Sigmoid)
# Model name
model_name = 'CNN5'
# Number of tests
num_test = 10

# Batch size is 50
batch_size = 50
# Number of epochs is 15
num_epochs = 15


# Create model, loss function, and optimizer for the model with ReLU activation
model = CNN5_R()
model.to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Initialize the lists
relu_loss_curve = []
relu_grad_curve = []

# Shuffle train generator at each epoch
for epoch in range(num_epochs):

    for i, (images, labels) in enumerate(train_generator):
        # Convert data to PyTorch tensors
        images = images.numpy()
        labels = labels.numpy()
        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass and compute loss
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        # Record training accuracy every 10 steps
        if (i+1) % 10 == 0:

            # Use model.fc1 for MLPs, model.conv1
            first_layer_weights = model.conv1.weight.data.cpu().numpy()
            grad_mag = np.linalg.norm(model.conv1.weight.grad.data.cpu().numpy())
            relu_grad_curve.append(grad_mag)
            # Record training loss
            relu_loss_curve.append(loss.item())


        # Print training loss every 100 steps
        if (i+1) % 100 == 0:
             print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Grad Mag: {:.4f}'
                  .format(epoch+1, num_epochs, i+1, len(train_data)//batch_size,
loss.item(), grad_mag))


# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_generator:
        # Convert data to PyTorch tensors
        images = images.numpy()
        labels = labels.numpy()
        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)
        images = images.to(device)
        labels = labels.to(device)

        # Compute predictions
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    test_acc = 100 * correct / total
    print('Accuracy of the model on the test images: {} %'.format(test_acc))

# Model with Sigmoid Function as Activation Function
# Create model, loss function, and optimizer for the model with ReLU activation
model = CNN5_S()
model.to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

sigmoid_loss_curve = []
sigmoid_grad_curve = []

# Shuffle train generator at each epoch
for epoch in range(num_epochs):

    for i, (images, labels) in enumerate(train_generator):
        # Convert data to PyTorch tensors
        images = images.numpy()
        labels = labels.numpy()
        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass and compute loss
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        # Record training accuracy every 10 steps
        if (i+1) % 10 == 0:

            # Use model.fc1 for MLPs, model.conv1
            first_layer_weights = model.conv1.weight.data.cpu().numpy()
            grad_mag = np.linalg.norm(model.conv1.weight.grad.data.cpu().numpy())
            sigmoid_grad_curve.append(grad_mag)

            # Record training loss
            sigmoid_loss_curve.append(loss.item())


        # Print training loss every 100 steps
        if (i+1) % 100 == 0:
             print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Grad Mag: {:.4f}'
                  .format(epoch+1, num_epochs, i+1, len(train_data)//batch_size,
loss.item(), grad_mag))


# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_generator:
        # Convert data to PyTorch tensors
        images = images.numpy()
        labels = labels.numpy()
        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)
        images = images.to(device)
        labels = labels.to(device)

        # Compute predictions
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        test_acc = 100 * correct / total
    print('Accuracy of the model on the test images: {} %'.format(test_acc))

import pickle

# create the dictionary object with the required key-value pairs
result_dict = {
    'name': model_name,
    'relu_loss_curve': relu_loss_curve,
    'sigmoid_loss_curve': sigmoid_loss_curve,
    'relu_grad_curve': relu_grad_curve,
    'sigmoid_grad_curve': sigmoid_grad_curve,
}

# save the dictionary object to a file
filename = 'part4_'+str(model_name)+'.pkl'
with open(filename, 'wb') as file:
    pickle.dump(result_dict, file)

# load the dictionary object from the file
with open(filename, 'rb') as file:
    loaded_dict = pickle.load(file)

# print the loaded dictionary object
print(loaded_dict)


import numpy as np
from matplotlib import pyplot as plt


# load the dictionary object from the file
with open('part4_MLP1.pkl', 'rb') as file:
    loaded_dict1 = pickle.load(file)

# load the dictionary object from the file
with open('part4_MLP2.pkl', 'rb') as file:
    loaded_dict2 = pickle.load(file)

# load the dictionary object from the file
with open('part4_CNN3.pkl', 'rb') as file:
    loaded_dict3 = pickle.load(file)

# load the dictionary object from the file
with open('part4_CNN4.pkl', 'rb') as file:
    loaded_dict4 = pickle.load(file)
```

```
# load the dictionary object from the file
with open('part4_CNN5.pkl', 'rb') as file:
    loaded_dict5 = pickle.load(file)

results = [loaded_dict1, loaded_dict2, loaded_dict3, loaded_dict4, loaded_dict5]

part4Plots(results, save_dir=r'data', filename='part4Plots')


### End of Question 4 ###
```

APPENDIX IV

```
### Question 5 ###

# Import PyTorch library and optimization module
import torch
import torch.optim as optim

# Import torchvision library and numpy
import torchvision
import numpy as np
import torchvision.transforms as transforms

# Import train_test_split from scikit-learn
from sklearn.model_selection import train_test_split

# Check if GPU is available, else use CPU
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# CIFAR-10 images have a size of 32x32


# Define a series of image transformations to be applied to the CIFAR-10 dataset
transform = transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
        torchvision.transforms.Grayscale()
])

# Load the CIFAR-10 dataset into a PyTorch data loader, applying the defined
transformations to it
train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True,
transform=transform)
```

```
# Split the training data into a training set and a validation set, with a split ratio
of 90:10
train_set, val_data = train_test_split(train_data, test_size=0.1, random_state=42)

# Create a PyTorch data loader for the training set
train_generator = torch.utils.data.DataLoader(train_set, batch_size=50, shuffle=True)

# Load the CIFAR-10 test dataset into a PyTorch data loader, applying the same
transformations as for the training data
test_data = torchvision.datasets.CIFAR10('./data', train=False, transform=transform)

# Create a PyTorch data loader for the test set
test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)

# Create a PyTorch data loader for the validation set
val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)


class CNN4_R(torch.nn.Module):
    def __init__(self):
        super(CNN4_R, self).__init__()
        # Conv. 3x3x16 Layer
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=
(3,3) , stride = 1, padding = 'valid')
        # ReLU layer
        self.relu1 = torch.nn.ReLU()
        # Conv. 3x3x8 Layer
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8,
kernel_size=(3,3), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu2 = torch.nn.ReLU()
        # Conv. 5x5x16 Layer
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16,
kernel_size=(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu3 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding=0)
        # Conv. 5x5x16 Layer
        self.conv4 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=
(5,5), stride = 1, padding = 'valid')
        # ReLU layer
        self.relu4 = torch.nn.ReLU()
        # MaxPool 2x2
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size= (2,2), stride = 2, padding =
0)
        # Prediction layer provides output with 10 classes
        self.prediction_layer = torch.nn.Linear(in_features= 16*4*4, out_features=10)
    def forward(self, x):
        # torch.Size([50, 1, 32, 32])
        x = self.conv1(x)
        # torch.Size([50, 16, 30, 30])
        x = self.relu1(x)
        x = self.conv2(x)
        # torch.Size([50, 8, 28, 28])
        x = self.relu2(x)
        x = self.conv3(x)
        # torch.Size([50, 16, 24, 24])
        x = self.relu3(x)
        x = self.maxpool1(x)
        # torch.Size([50, 16, 12, 12])
        x = self.conv4(x)
        x = self.relu4(x)
        # torch.Size([50, 16, 8, 8])
        x = self.maxpool2(x)
        # torch.Size([50, 16, 4, 4])
        x = x.view(50 ,16*4*4)
        # torch.Size([50, 256])
        x = self.prediction_layer(x)
        return x


# Model name
model_name = 'CNN4_R'

# Batch size is 50
batch_size = 50
# Number of epochs is 20
num_epochs = 20

lr_list = [0.1, 0.01, 0.001]
train_loss_complete = []
val_acc_complete = []

# Train the model 3 times for different learning rates (0.1, 0.01, 0.001)
for x in range(len(lr_list)):

  model = CNN4_R()
  model.to(device)
  # Determine the loss function
  criterion = torch.nn.CrossEntropyLoss()
  # Using SGD optimizer instead of the Adam optimizer
  optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum = 0)

  # Initializing lists
  train_loss = []
  val_acc = []
```

```python
for epoch in range(num_epochs):

    for i, (images, labels) in enumerate(train_generator):

        # Convert data to PyTorch tensors
        images = images.numpy()
        labels = labels.numpy()

        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)

        images = images.to(device)
        labels = labels.to(device)

        # Forward pass and compute loss
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        # Record training accuracy every 10 steps
        if (i+1) % 10 == 0:

            # Record training loss
            train_loss.append(loss.item())

            with torch.no_grad():
                total = 0
                correct = 0

                #outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                train_accuracy = 100 * correct / total

                # Record validation accuracy every 10 steps
                total = 0
                correct = 0
                for images, labels in val_generator:
                    images = images.to(device)
                    labels = labels.to(device)
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
```

```
                  val_accuracy = 100 * correct / total
                  val_acc.append(val_accuracy)

      # To find out, at which step test accuracy converges to some value
      print('Val. Acc.: ', val_accuracy)
      print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
            .format(epoch+1, num_epochs, i+1, len(train_data)//batch_size,
loss.item()))


  print(len(train_loss))
  # Test the model
  model.eval()
  with torch.no_grad():
      correct = 0
      total = 0
      for images, labels in test_generator:

          # Convert data to PyTorch tensors
          images = images.numpy()
          labels = labels.numpy()

          images = torch.from_numpy(images)
          labels = torch.from_numpy(labels)

          images = images.to(device)
          labels = labels.to(device)

          # Compute predictions
          outputs = model(images)
          _, predicted = torch.max(outputs.data, 1)
          total += labels.size(0)
          correct += (predicted == labels).sum().item()
      test_acc = 100 * correct / total
      print('Accuracy of the model on the test images: {} %'.format(test_acc))

  train_loss_complete.append(train_loss)
  val_acc_complete.append(val_acc)

import pickle

# create the dictionary object with the required key-value pairs
result_dict = {

    'name': model_name,
    'loss_curve_1': train_loss_complete[0],
    'loss_curve_01': train_loss_complete[1],
    'loss_curve_001': train_loss_complete[2],
    'val_acc_curve_1': val_acc_complete[0],
    'val_acc_curve_01': val_acc_complete[1],
    'val_acc_curve_001': val_acc_complete[2]

}

# save the dictionary object to a file
filename = 'part5_'+str(model_name)+'.pkl'
with open(filename, 'wb') as file:
    pickle.dump(result_dict, file)

# load the dictionary object from the file
with open(filename, 'rb') as file:
    loaded_dict = pickle.load(file)

# print the loaded dictionary object
print(loaded_dict)


# Model name
model_name = 'CNN4_R'

# Batch size is 50
batch_size = 50
# Number of epochs is 20
num_epochs = 30

train_loss_complete = []
val_acc_complete = []

# Now, learning rate is changed to 0.01 after 8th epoch.
# Learning rate is decreased to conitnue the process of increasing the validation
accuracy
# Create model, loss function, and optimizer
model = CNN4_R()
model.to(device)
# Determine the loss function
criterion = torch.nn.CrossEntropyLoss()
# Using SGD optimizer instead of the Adam optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum = 0)

# Initializing lists
train_loss = []
val_acc = []

for epoch in range(num_epochs):

  # At epoch = 10, lr = 0.1 increases the validation accuracy no more
  # lr is changed to 0.01
  if(epoch == 10):
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum = 0)
```

```python
for i, (images, labels) in enumerate(train_generator):

    # Convert data to PyTorch tensors
    images = images.numpy()
    labels = labels.numpy()

    images = torch.from_numpy(images)
    labels = torch.from_numpy(labels)

    images = images.to(device)
    labels = labels.to(device)

    # Forward pass and compute loss
    outputs = model(images)
    loss = criterion(outputs, labels)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()


    # Record training accuracy every 10 steps
    if (i+1) % 10 == 0:

        # Record training loss
        train_loss.append(loss.item())

        with torch.no_grad():
            total = 0
            correct = 0

            #outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            train_accuracy = 100 * correct / total

            # Record validation accuracy every 10 steps
            total = 0
            correct = 0
            for images, labels in val_generator:
                images = images.to(device)
                labels = labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
            val_accuracy = 100 * correct / total
            val_acc.append(val_accuracy)
```

```
  # To find out, at which step test accuracy converges to some value
  print('Val. Acc.: ', val_accuracy)
  print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
        .format(epoch+1, num_epochs, i+1, len(train_data)//batch_size, loss.item()))


  # Test the model
  model.eval()
  with torch.no_grad():
      correct = 0
      total = 0
      for images, labels in test_generator:

          # Convert data to PyTorch tensors
          images = images.numpy()
          labels = labels.numpy()

          images = torch.from_numpy(images)
          labels = torch.from_numpy(labels)

          images = images.to(device)
          labels = labels.to(device)


  train_loss_complete.append(train_loss)
  val_acc_complete.append(val_acc)


import pickle

# create the dictionary object with the required key-value pairs
result_dict = {

    'name': model_name,
    'loss_curve_1': train_loss_complete[0],
    'loss_curve_01': train_loss_complete[1],
    'loss_curve_001': train_loss_complete[2],
    'val_acc_curve_1': val_acc_complete[0],
    'val_acc_curve_01': val_acc_complete[1],
    'val_acc_curve_001': val_acc_complete[2]

}

# save the dictionary object to a file
filename = 'part5_'+str(model_name)+'.pkl'
with open(filename, 'wb') as file:
    pickle.dump(result_dict, file)

# load the dictionary object from the file
```

```
with open(filename, 'rb') as file:
    loaded_dict = pickle.load(file)

# print the loaded dictionary object
print(loaded_dict)

part5Plots(loaded_dict, save_dir=r'data', filename='part5Plots_lr01_lr001')




# Model name
model_name = 'CNN4_R'

# Batch size is 50
batch_size = 50
# Number of epochs is 20
num_epochs = 30

train_loss_complete = []
val_acc_complete = []

# Now, learning rate is changed to 0.01 after 8th epoch.
# Learning rate is decreased to conitnue the process of increasing the validation
accuracy
# Create model, loss function, and optimizer
model = CNN4_R()
model.to(device)
# Determine the loss function
criterion = torch.nn.CrossEntropyLoss()
# Using SGD optimizer instead of the Adam optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum = 0)

# Initializing lists
train_loss = []
val_acc = []

for epoch in range(num_epochs):

  # At epoch = 10, lr = 0.1 increases the validation accuracy no more
  # lr is changed to 0.01
  if(epoch == 10):
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum = 0)
  # At epoch = 15, lr = 0.01 increases the validation accuracy no more
  # lr is changed to 0.001
  elif(epoch ==15):
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum = 0)

  for i, (images, labels) in enumerate(train_generator):

      # Convert data to PyTorch tensors
```

```
        images = images.numpy()
        labels = labels.numpy()

        images = torch.from_numpy(images)
        labels = torch.from_numpy(labels)

        images = images.to(device)
        labels = labels.to(device)

        # Forward pass and compute loss
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        # Record training accuracy every 10 steps
        if (i+1) % 10 == 0:

            # Record training loss
            train_loss.append(loss.item())

            with torch.no_grad():
                total = 0
                correct = 0

                #outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                train_accuracy = 100 * correct / total

                # Record validation accuracy every 10 steps
                total = 0
                correct = 0
                for images, labels in val_generator:
                    images = images.to(device)
                    labels = labels.to(device)
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
                val_accuracy = 100 * correct / total
                val_acc.append(val_accuracy)

    # To find out, at which step test accuracy converges to some value
    print('Val. Acc.: ', val_accuracy)
```

```
  print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
        .format(epoch+1, num_epochs, i+1, len(train_data)//batch_size, loss.item()))


  # Test the model
  model.eval()
  with torch.no_grad():
      correct = 0
      total = 0
      for images, labels in test_generator:

          # Convert data to PyTorch tensors
          images = images.numpy()
          labels = labels.numpy()

          images = torch.from_numpy(images)
          labels = torch.from_numpy(labels)

          images = images.to(device)
          labels = labels.to(device)


  train_loss_complete.append(train_loss)
  val_acc_complete.append(val_acc)


import pickle

# create the dictionary object with the required key-value pairs
result_dict = {

    'name': model_name,
    'loss_curve_1': train_loss_complete[0],
    'loss_curve_01': train_loss_complete[1],
    'loss_curve_001': train_loss_complete[2],
    'val_acc_curve_1': val_acc_complete[0],
    'val_acc_curve_01': val_acc_complete[1],
    'val_acc_curve_001': val_acc_complete[2]

}

# save the dictionary object to a file
filename = 'part5_'+str(model_name)+'.pkl'
with open(filename, 'wb') as file:
    pickle.dump(result_dict, file)

# load the dictionary object from the file
with open(filename, 'rb') as file:
    loaded_dict = pickle.load(file)
```

```python
# print the loaded dictionary object
print(loaded_dict)

part5Plots(loaded_dict, save_dir=r'data', filename='part5Plots_lr01_lr001_lr_0001')

### End of Question 5 ###
```