Süleyman Yasin Peker
2305191

EE449 – Computational Intelligence

Homework 1 - Training Artificial Neural Network

Part1. Basic Concepts

1.1. Which Function?

The ANNs classifier with cross-entropy loss approximates the true conditional probability distribution of the targets which of those input features are given. The conditional probability distribution is shown as $P(y|x)$. The term entropy can be described as the amount of uncertainty of an event. The mathematical representation of the cross-entropy loss is as follows:

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

i: Label index

y_i: True label with index i

$\hat{y_i}$: Predicted label with index i

The cross-entropy loss approximates the true conditional probability distribution by penalizing the wrong decisions. It enhances the model to predict high probabilities for true labels and low probabilities for false labels. By doing so the cross-entropy loss reduces and as the cross-entropy loss decreases, the approximation of the true probability distribution becomes better. This gives better classification results. To make the classifications better with high probability distributions, the cross-entropy loss is used. That's why it is important to ANNs.

1.2. Gradient Computation

By using the SGD, the update rule can be defined as follows:

$$w_{k+1} = w_k - \gamma \nabla_{w_k} L$$

Then, isolate the gradient loss term from the above equation.

$$\nabla_{w_k} L = \frac{w_k - w_{k+1}}{\gamma}$$

By using the chain rule, the gradient loss with respect to the weights at step k can be written in terms of the gradient loss with respect to the weights at step k+1.

$$\nabla_w L|_{w=w_k} = \nabla_{w_{k+1}} L \times \nabla_{w_k w_{k+1}} w_{k+1}$$

After putting the calculated gradient loss term into the gradient loss with respect to the weights at step k, we obtain the following equation:

$$\nabla_w L|_{w=w_k} = \nabla_{w_{k+1}} L \times \frac{w_k - w_{k+1}}{\gamma}$$

1.3. Some Training Parameters and Basic Parameter Calculations

1. The batch size is the number of samples that the model is trained on for each time. While the specified batch is being processed, the model is not updated.

The number of epochs is the number that the whole training dataset is processed. In other words, all of the training dataset is passed through the model.

2. The number of batches can be calculated by the following equation:

$$Number\ of\ batches = \frac{N}{B}$$

Total number of samples are divided into batches and the number of batches can be found by dividing the number of samples into the batch size.

3. Number of batches are found by the following formula as it was calculated in the previous question:

$$Number\ of\ batches = \frac{N}{B}$$

This number of batches specifies the iteration number of training for each epoch. So, by using the number of batches, the number of SGD iterations can be calculated by the following equation:

$$Number\ of\ SGD\ iterations = E * \frac{N}{B}$$

In other words, for E epoch, the total number of iterations are found by multiplication of number of epochs and number of batches.

1.4. Computing Number of Parameters of ANN Classifiers

1. The number of parameters of an MLP classifier can be calculated by the sum of the parameters in both weights and biases for all layers. Firstly, the number of parameters in the first layer can be calculated as follows:

$$Number\ of\ parameters\ of\ the\ first\ layer = H_1 \times D_{in} + H_1$$

The size of the of the first layer is the multiplication of $D_{in}\ and\ H_1$. Also, there is a bias matrix which has the size of $H_1 \times 1$. By adding these two gives us the number of parameters of the first layer

The rest of the hidden layers have the same number of parameters as the first layer because all of them are the size of H_1. So, multiplication of the number of parameters of the first layer with the number of hidden layers gives the number of parameters of the hidden layers as follows:

$$Number\ of\ parameters\ of\ the\ hidden\ layers = K \times (H_1 \times D_{in} + H_1)$$

Finally, the number of parameters of the output layer can be calculated by the multiplication of $D_{out}\ and\ H_k$ which gives the parameters of the weight and D_{out} is the size of the bias. The following equation gives the resultant parameters of the output layer:

$$Number\ of\ parameters\ of\ the\ output\ layer = H_k \times D_{out} + H_k$$

So, the following equation gives the total number of parameters of the MLP:

$$\begin{aligned}&Number\ of\ parameters\ of\ the\ MLP \\ &\qquad = (H_1 \times D_{in} + H_1) + K \times (H_1 \times D_{in} + H_1) + H_k \times D_{out} + H_k\end{aligned}$$

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

Each convolutional layer has C_{k-1} input channels and C_k output channels. Also, the spatial size of each convolutional layer is $H_k \times W_k$. Moreover, each convolutional layer has a bias term. Then, by adding up these parameters, we obtain the following formula:

$$Number\ of\ Parameters\ of\ a\ Convolutional\ Layer = (H_k \times W_k \times C_{k-1} \times C_k) + C_k$$

The input layer has neither bias nor weight. That's why the number of parameters are calculated starting from the first layer. The number of parameters of the first convolutional layer can be calculated as follows:

$$Number\ of\ Parameters\ of\ the\ First\ Convolutional\ Layer = (H_1 \times W_1 \times C_{in} \times C_1) + C_1$$

So, by adding up these two formulas, we can find the total number of parameters of a CNN.

$$Total\ Number\ of\ Parameters\ of\ a\ CNN$$
$$= [(H_1 \times W_1 \times C_{in} \times C_1) + C_1] + \sum_{2}^{k} [((H_k \times W_k \times C_{k-1} \times C_k) + C_k]$$

The first term corresponds to the number of parameters of the first convolutional layer, and the sum term corresponding to the rest of the layers from layer 2 to layer k.

Part2. Implementing a Convolutional Layer with NumPy

2.1. Convolutional Neural Networks (CNNs) are widely used at the center of deep learning algorithms. The previous networks, called traditional neural networks, are less effective than CNNs because of their input shape restrictions. In order to use these traditional neural networks, the input size should be a constant number. In addition, traditional neural networks have many parameters to be trained. As the number of parameters increases, the effectiveness of the network decreases. So, it is hard to train a traditional neural network.

On the other hand, CNNs have a better approach to these problems. It is not restricted to using a fixed input shape while using CNNs. In other words, the shape of the input can be arbitrary. Furthermore, the number of parameters in the CNNs is decreased. That's why it is easier to train a CNN when compared to a traditional neural network.

Especially for computer vision and image processing tasks, CNNs are used a lot. The main reason is that CNNs can extract features from the input. These features are edges, corners, patterns, and textures. Object detection, segmentation, and image classification operations can be done effectively using these feature extraction abilities. To conclude, using CNNs in image processing tasks is important since CNNs can extract features from inputs efficiently.

2.2. Kernels are filters that can extract features from the input image. A kernel is a matrix in which the weights are involved. Generally, the kernel size is chosen to be smaller than the input size in order to extract local features from the input. The kernel is convolved with the input image and gives an output smaller than the input image. The stride value specifies the movement of the kernel. For example, if the stride is selected as 1, then the kernel, or filter, moves one by one through the input matrix, and each movement kernel is performed a dot product operation with the input matrix. By doing so, high-level features such as patterns, edges, and corners can be extracted from the input.

The sizes of a kernel depend on the application. For example, a smaller kernel should be used to extract local features from the input. On the other hand, to extract larger features, a larger kernel can be used. The size of the kernel consists of two parameters, namely kernel height and kernel width. Kernel height is the number of rows in the kernel, and the kernel width is the number of columns in the kernel. Generally, kernels are chosen as a square matrix, but it is not mandatory. Rectangular kernels can be used as well.

2.3. After the convolutional layer, an output image is generated. This image has 5 rows and 8 columns. Each row corresponds to the same number class, and each column corresponds to a different kernel. The convolutional layer tried to extract features from the given input, and this resulted in 8 different channels for each number. But as it can be seen from the image, the pattern of the number is not understood yet to decide which number class is detected. That's why the convolutional layer itself is not sufficient. According to different kernels, different outputs are obtained. It can be concluded that outputs of the same kernel looks like each other, and outputs of different kernels are different from each other.

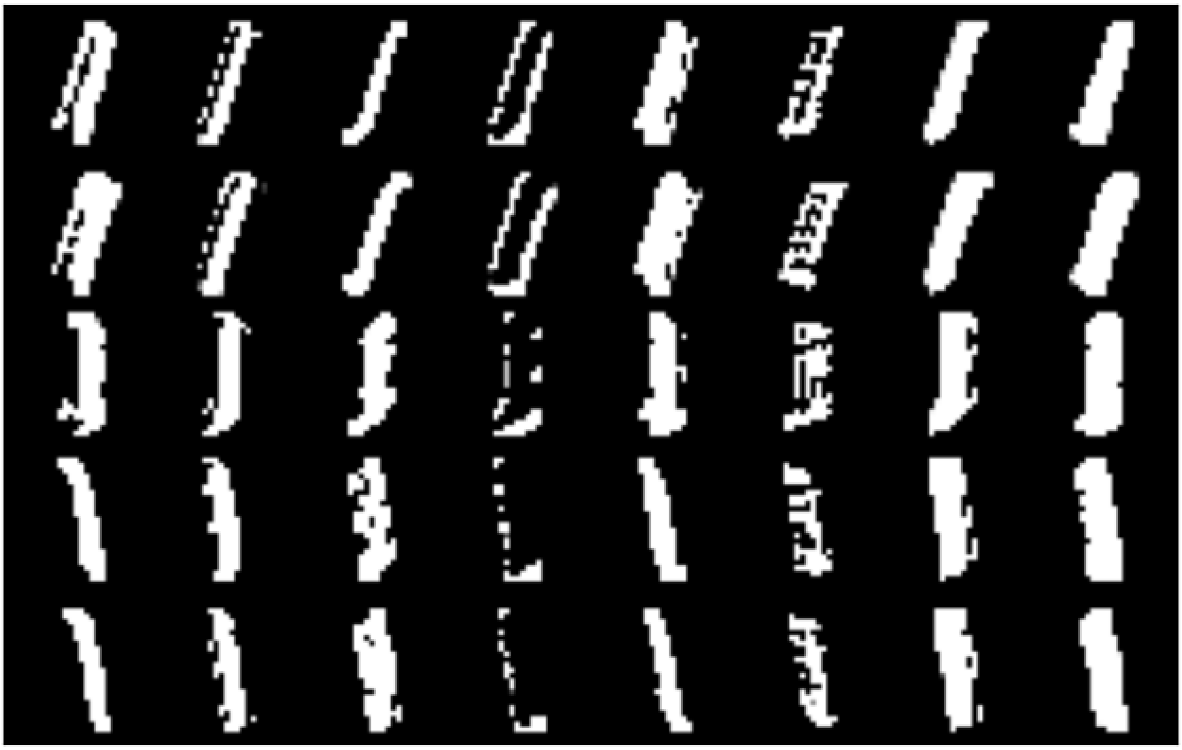
Figure 1. Output of the Convolutional Layer

2.4. The convolutional layer consists of eight 4x4 kernels. Each kernel has its own weights. So, after the convolution operation, output of the convolutional layer will give us 8 different images or matrices which are called channels. In the output image, rows correspond to different batches and columns corresponds to different kernels. When we examine the same column, almost all of the number in a column are similar to each other. The main reason of this similarity comes from the same kernel is applied to all of these numbers. Even though they are different numbers, the output channel looks like similar. In other words, kernel weights are important fact that affect the output of the convolutional layer. Output channels of the same kernel can look like to each other.

2.5. As it was stated in the previous question, the convolutional layer consists of eight 4x4 kernels, and each row corresponds to the same number class, but the output channels are not similar. The main reason of this issue is that different kernels have different weights, and the input weights are the same for the same row. So, for each convolution operation gives different output. For the same number, different kernels are applied. Even though the number does not change, the output channel looks different than the others.

2.6. For feature extraction, using convolutional layers are important but using only one type of kernel will give us wrong intuitions about the input image. That's why using more kernels with different weight could be better approach to extract patterns from the input images. Also, by using only one convolutional layer is not sufficient. Using more than one convolutional layer could give better results. Furthermore, after examining the output image, it can be understood that using only one convolutional layer cannot extract complex patterns and high-level features. This type of networks can be described as shallow networks and to make feature extraction of these networks better, deeper neural networks can be used. Deep neural networks can be created by using multiple convolutional layers that are stacked top on each other. In this way, more complex features can be extracted.

Part3. Experimenting ANN Architectures

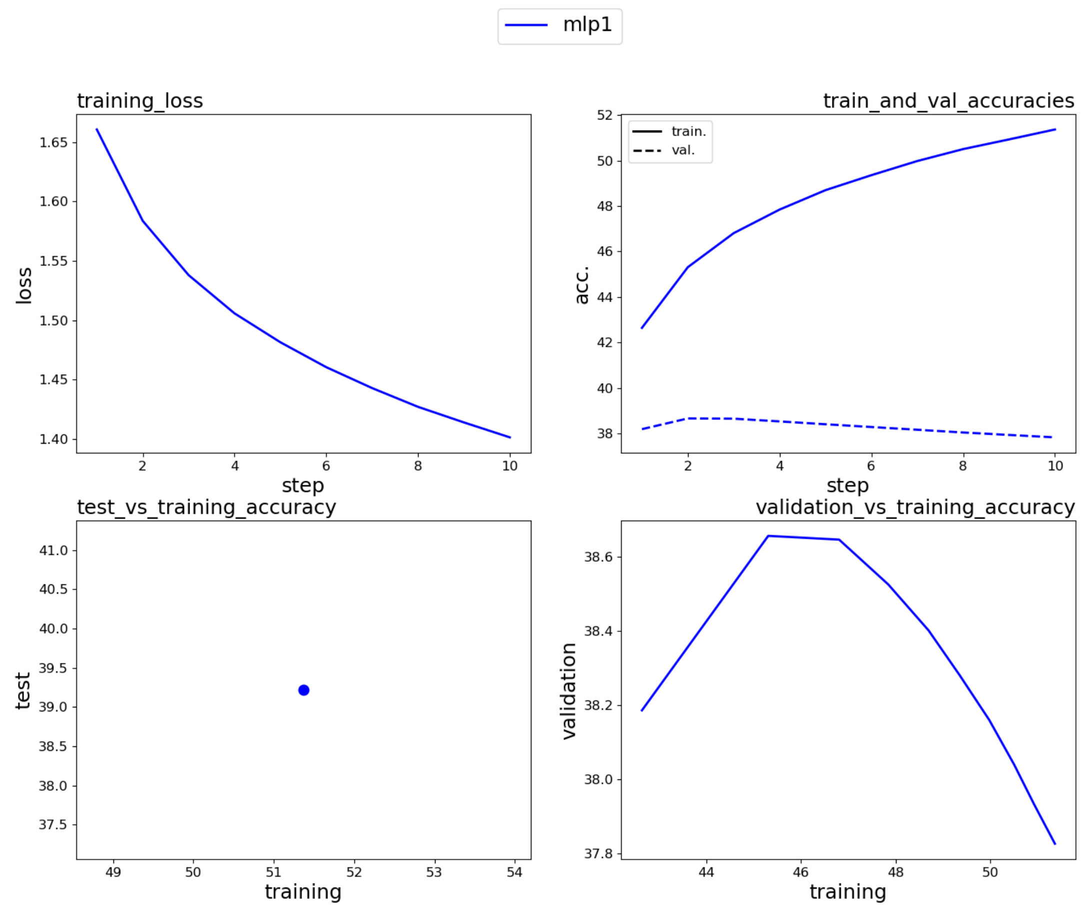

Figure 2. Training Plots of mlp1

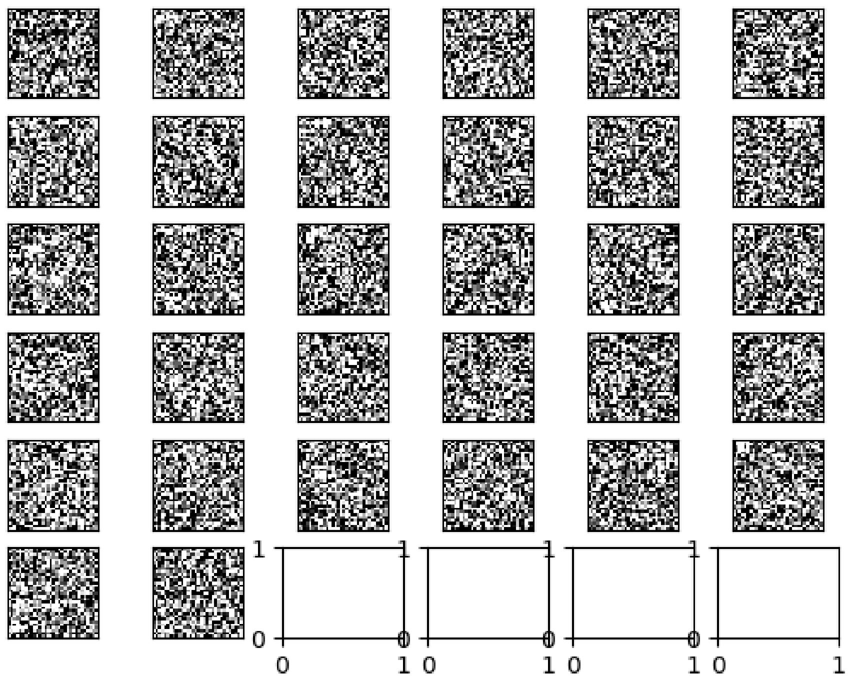

Figure 3. Initial Weights of the Model mlp1

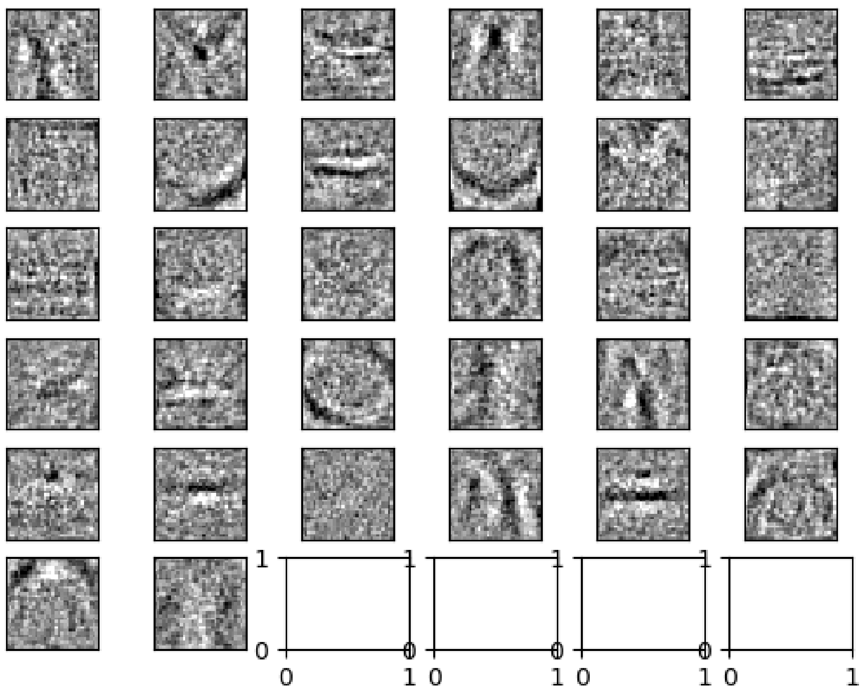

Figure 4. Final Weights of the Model mlp1

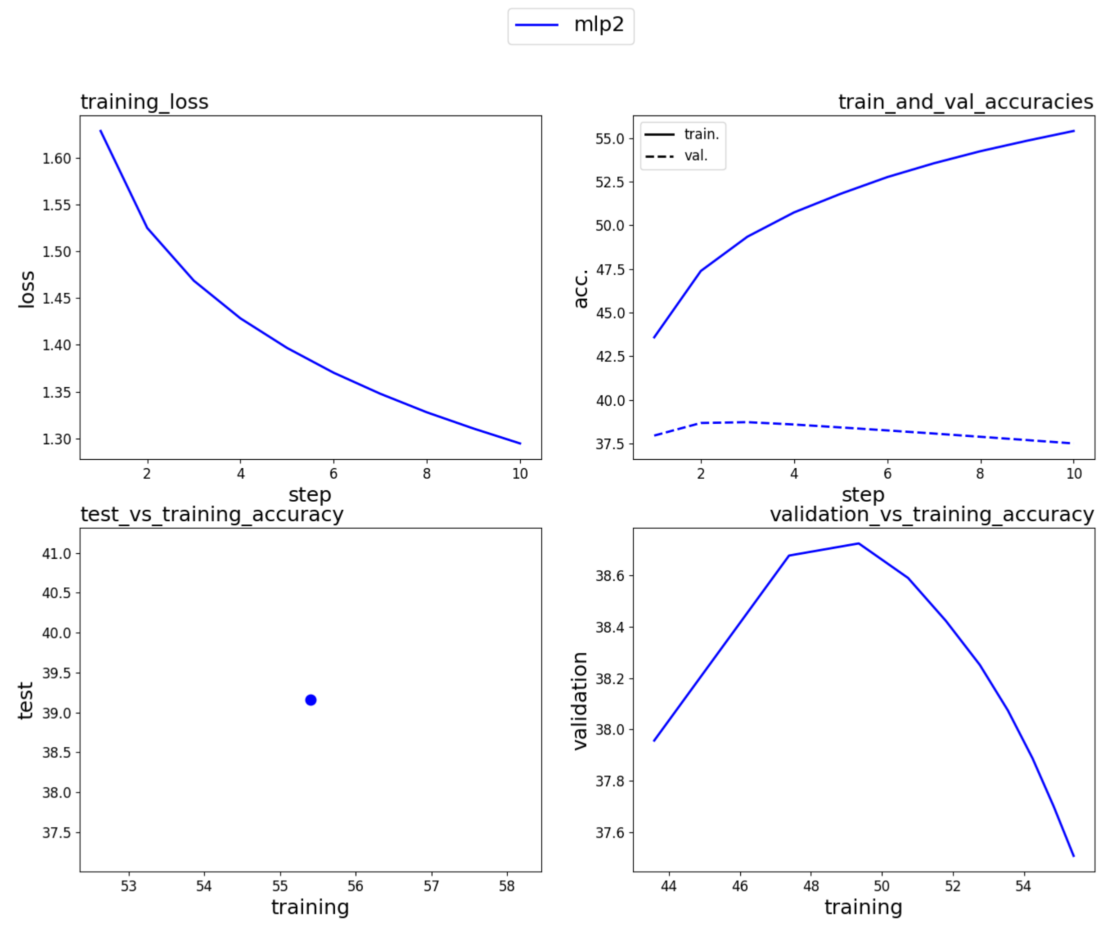

Figure 5. Training Plots of mlp2

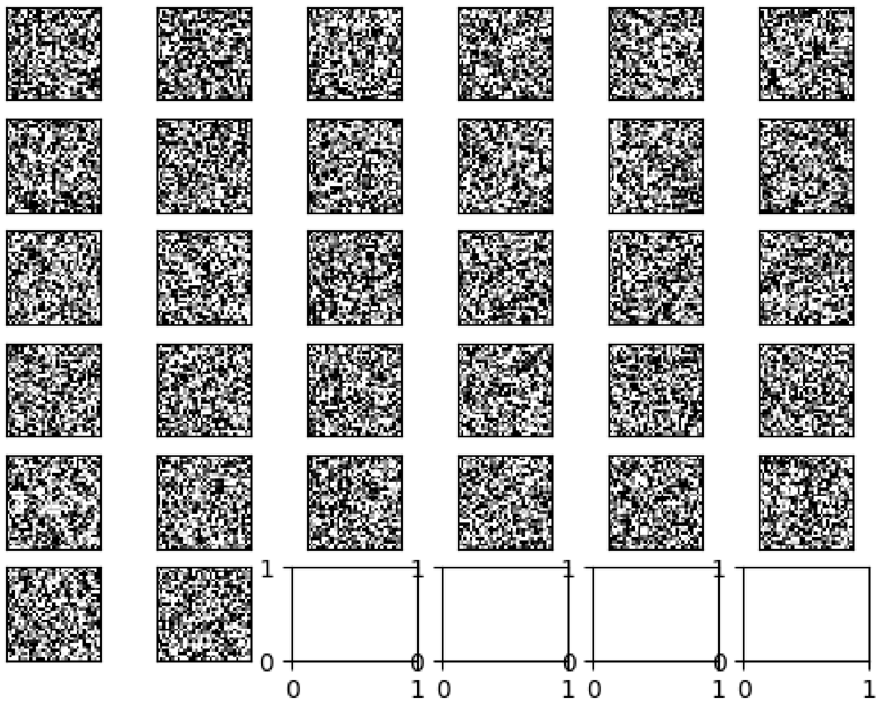

Figure 6. Initial Weights of the Model mlp2

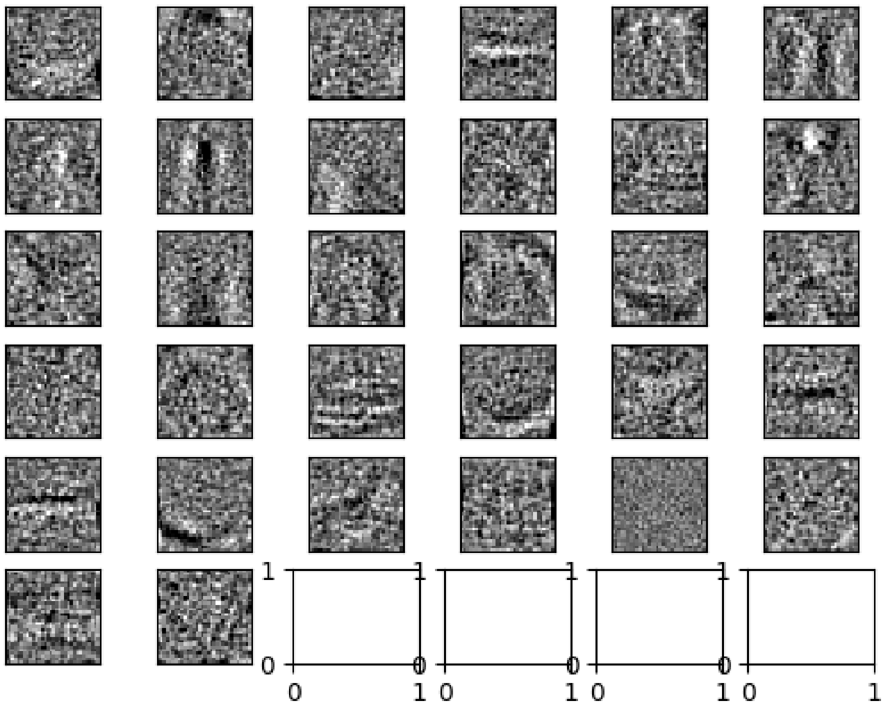

Figure 7. Final Weights of the Model mlp2

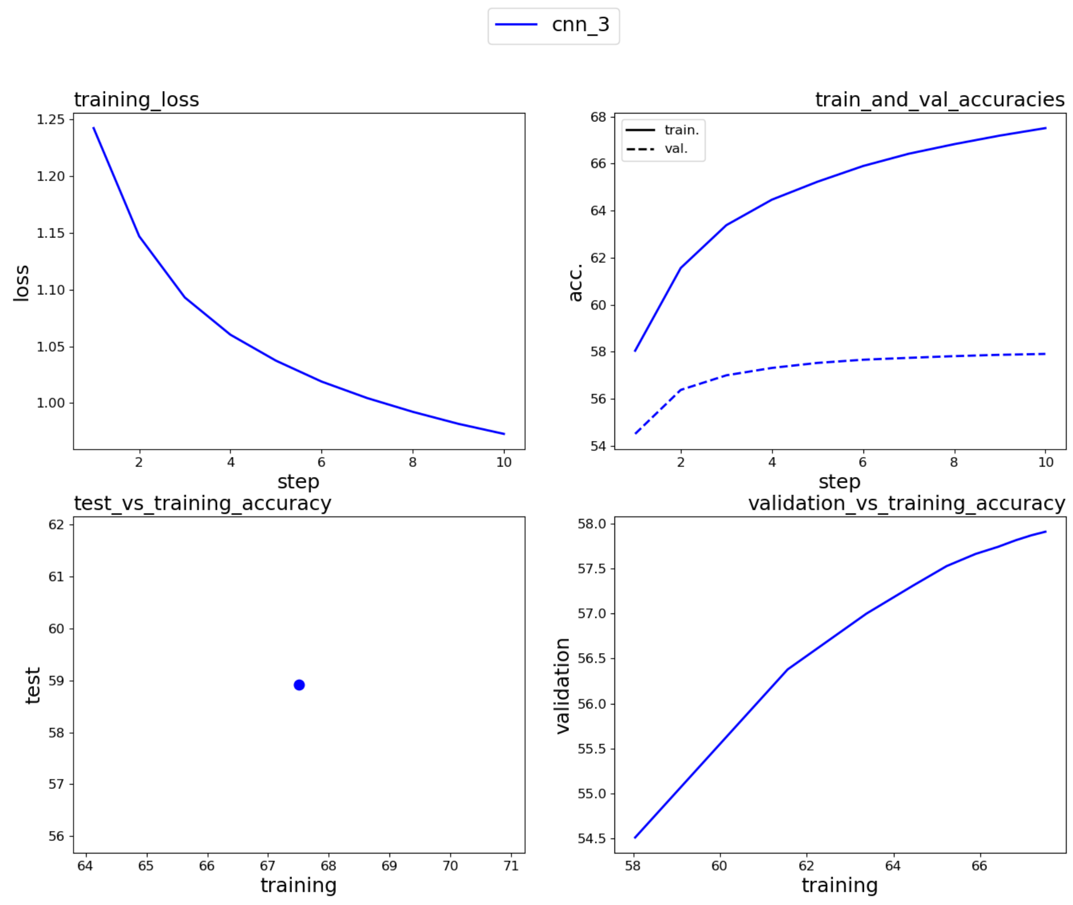

Figure 8. Training Plots of cnn_3

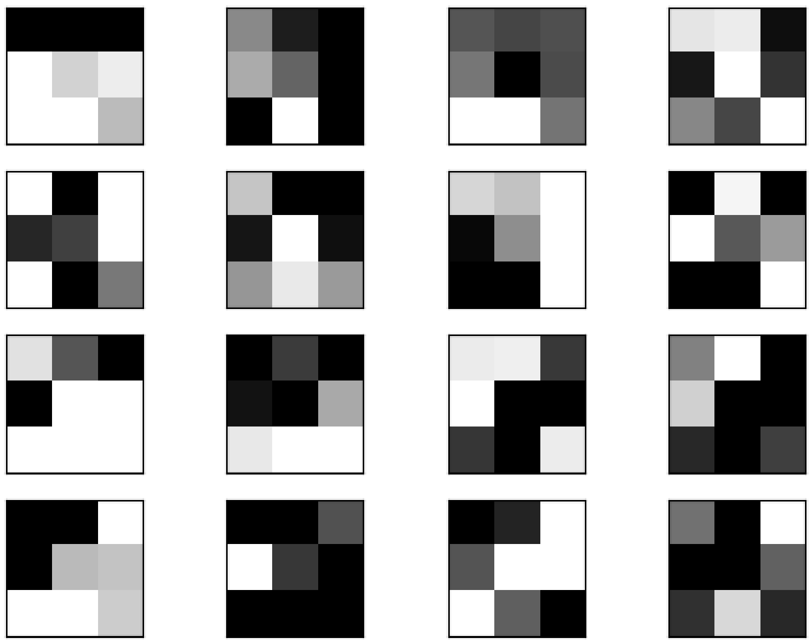
Figure 9. Initial Weights of the Model cnn_3

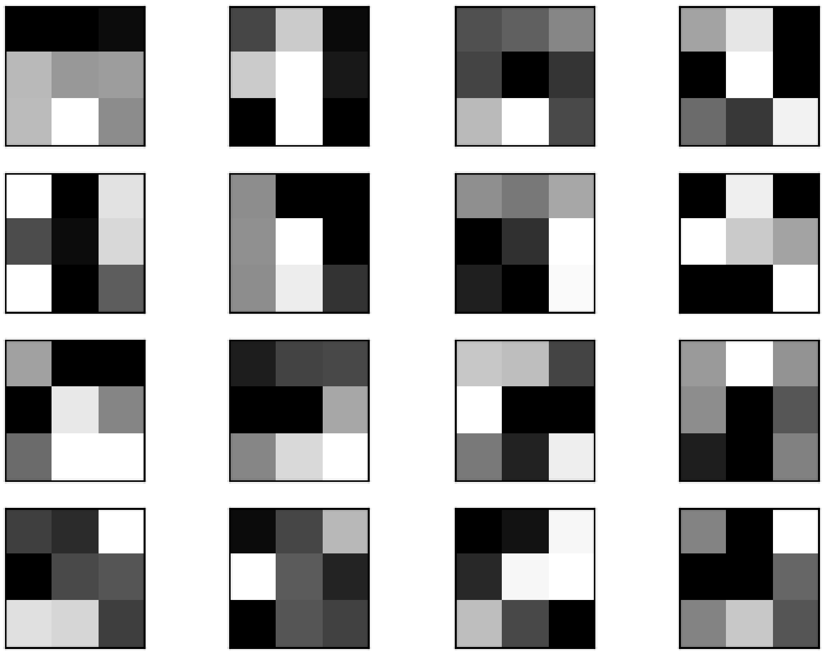
Figure 10. Final Weights of the Model cnn_3

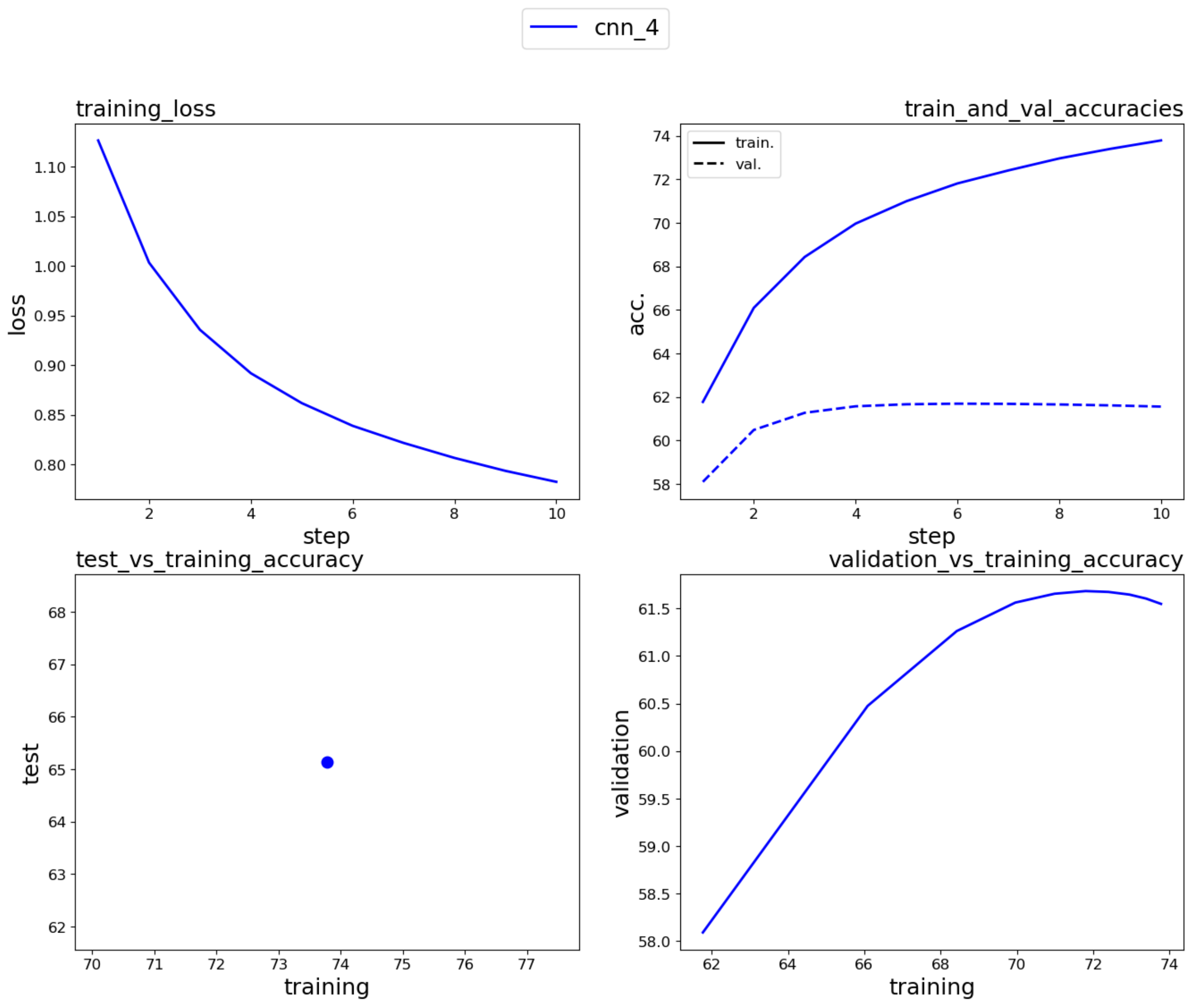

Figure 11. Training Plots of cnn_4

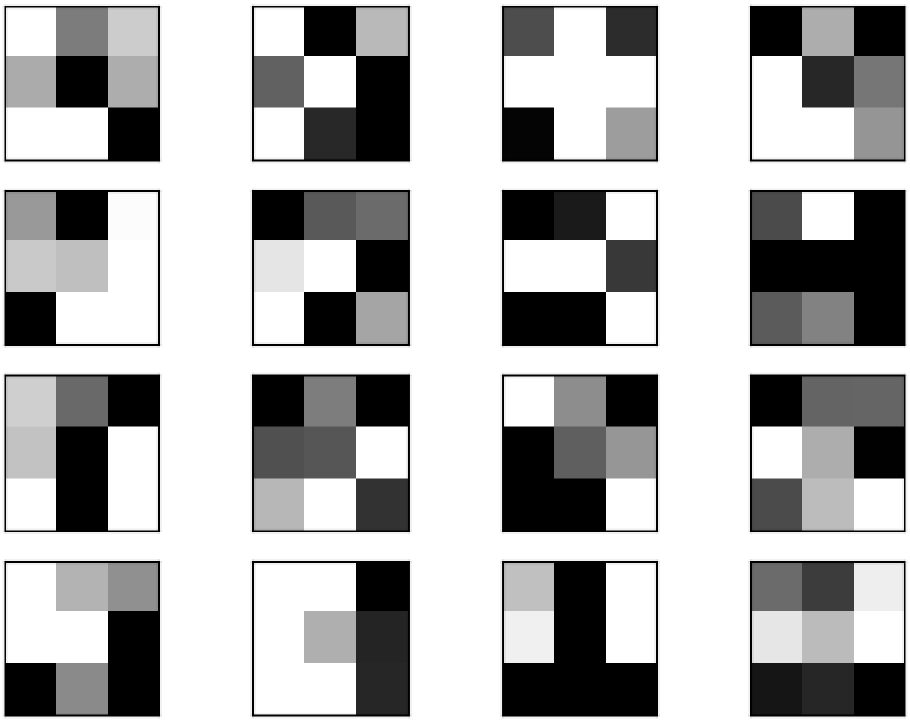
Figure 12. Initial Weights of the Model cnn_4

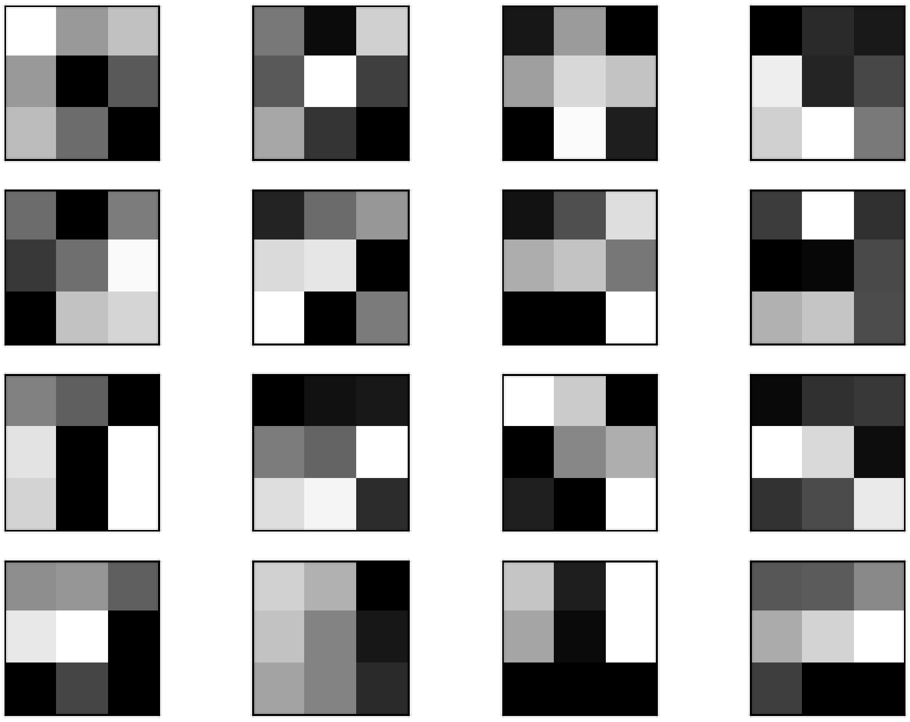
Figure 13. Final Weights of the Model cnn_4

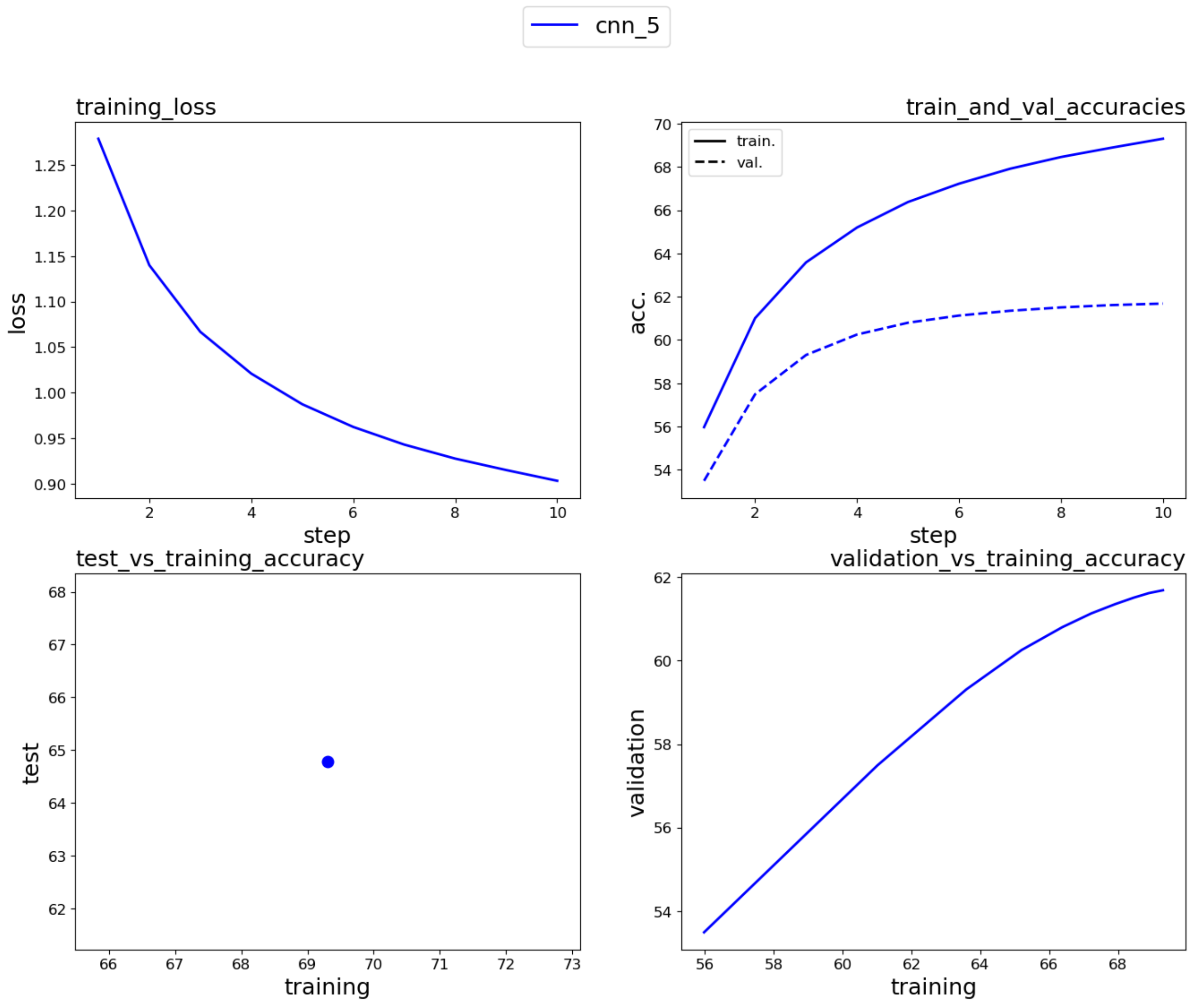

Figure 14. Training Plots of cnn_5

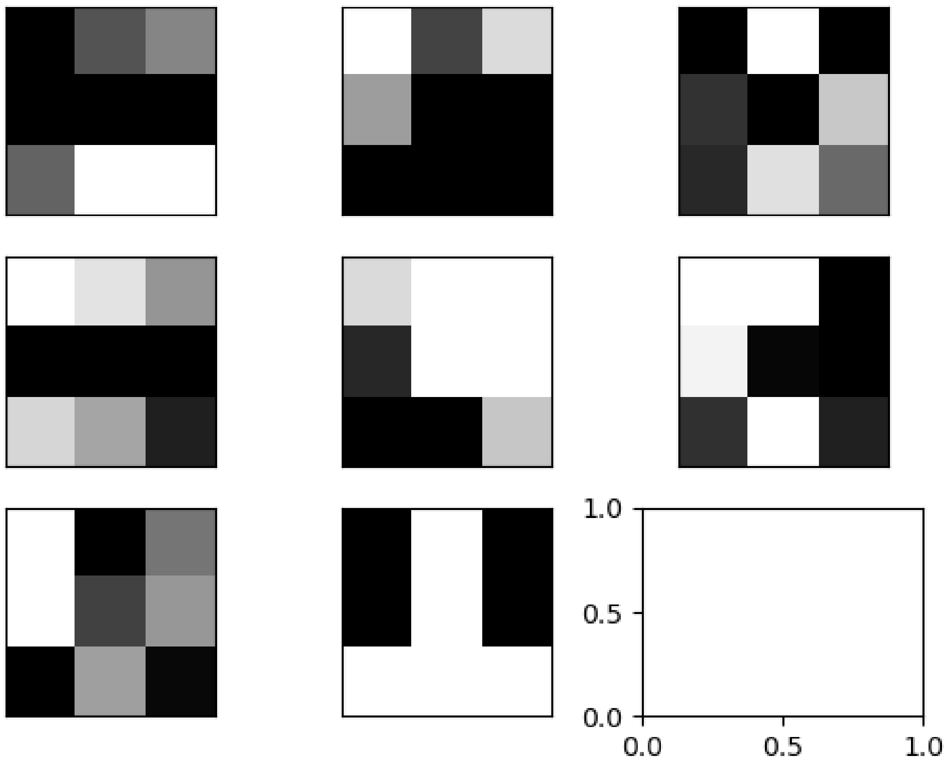

Figure 15. Initial Weights of the Model cnn_5

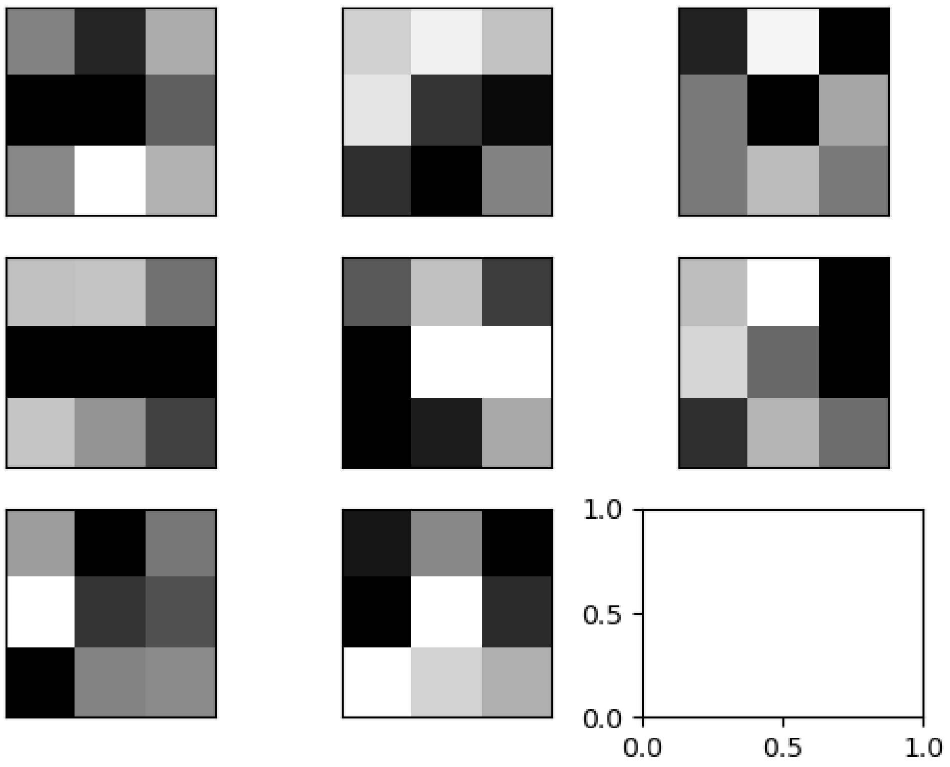

Figure 16. Final Weights of the Model cnn_5

Süleyman Yasin Peker
2305191

1. A classifier's generalization performance measures how effectively a classifier can correctly categorize new data that wasn't used during the training phase. In other words, it evaluates how well the classifier can apply the data it discovered from the training set to new, unexplored data.

A classifier with strong generalization performance may correctly categorize new data, even when different from the training data. On the other hand, the classifier may overfit the training data and perform poorly on new data if it has poor generalization performance.

Several metrics can be used to assess a classifier's generalization ability, including accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve (AUC-ROC). These metrics can be used to compare several classifiers' performance and ascertain how well the classifier works on new data.

2. The validation vs. training accuracy plot can give a better approach to inspect the generalization performance of the models since the training accuracy of the model most of the time increases, but the validation accuracy does not. The main reason for this issue is that the model starts to overfit the training data, especially for a high number of epochs and many runs that make the model overfit the data. That's why even though the model's training accuracy increases, the model's validation accuracy decreases. In addition to the validation accuracy vs. training accuracy plot, other types of plots can be used to inspect the model's generalization performance. These methods include Confusion Matrix, Prevision-Recall Curve, ROC Curve (Receiver Operating Characteristic), and Learning Curve. Still, in our case, we plotted the validation accuracy vs. training accuracy, training loss, and best test performance curves.

3. The generalization performance of the architectures can be compared by inspecting the validation accuracy vs. training accuracy plots. As we examine these plots, we can see that the multilayer perceptron architectures do not show a good generalization performance. Both mlp1 and mlp2 architectures have shown similar validation accuracy vs. training accuracy plots. In both cases, the training accuracy of the architectures increases. The training accuracies of the architectures started from 42 levels and rose to 52 levels.

On the other hand, the validation accuracies of the architectures remained at almost the same level, which is 38. Furthermore, as the training number increases, the validation accuracy of the architectures starts to decrease after a certain point which shows that these two architectures, mlp1, and mlp2 are not good at generalization. They overfit the data, and even though their training accuracies increased and their training losses decreased, the validation accuracies of these architectures show that they did not learn the pattern of the data correctly.

Regarding the CNN architectures, cnn_3, cnn_4, and cnn_5 architectures showed similar performances in terms of generalization. When I examine the training losses of these CNN architectures, the training loss of these architectures tends to decrease in time. If we increase the number of runs, some of the architectures will give a better performance. In addition to the training losses, the validation accuracy vs. training accuracy plots shows that all of these CNN architectures are good at generalization. The cnn_3 architecture had 58 training accuracy and 54.5 validation accuracy at the beginning of the training. Still, after the training finished, the training accuracy of the cnn_3 architecture was 67, and the validation accuracy of the model was 57, which indicates that the model learned the pattern of the data and increased the accuracy of the model.

Moreover, the cnn_4 architecture showed a similar training loss curve and validation accuracy vs. training accuracy curve with a slight difference. In this architecture, after step 4, the validation accuracy of the model has not improved much, which shows that model started to overfit the data.

The generalization of the architecture is about to be decreased since the training accuracy of the architecture tends to increase in time, but the validation accuracy did not. Lastly, the cnn_5 architecture showed one of the best generalization performances among the other architectures. The main reason for this conclusion is that the validation accuracy vs. training accuracy plot has a similar shape. In other words, as the training accuracy increases, the validation accuracy increases too, which makes the model prone to different data types since it has a better generalization performance.

4. The quantity of learnable weights that a machine learning model must modify during training is the number of parameters in the model. Generally speaking, a model with more parameters can represent more complicated functions, resulting in greater performance on the training set of data. This may or may not lead to improved generalization performance on unknown data.

In reality, overfitting, where the model is too closely adapted to the training data and performs poorly on new, unforeseen data, can result from having too many parameters. This is due to the possibility that the model is catching noise or peculiarities in the training data that do not transfer to new data. This is especially problematic when there are many more parameters than training data.

When we examine the architectures, the cnn_5 architecture was one of the best-performing models. One of the main reasons for this result is that there are several more parameters in this architecture, and in this way, it can learn the pattern better. On the other hand, if more than the number of parameters in an architecture is needed, the architecture can show wrong classification and generalization performances. So, the optimum number of parameters should be selected to train the model with good classification and generalization performances.

5. The number of layers in a machine learning model architecture is called the model's depth. In general, but only sometimes, a deeper architecture can improve performance on both the training and test data.

A deeper architecture has the benefit of being able to capture more abstract and complicated properties, which can be helpful when learning hierarchical representations of the data. As a result, the model can develop its ability to distinguish between classes, resulting in improved performance on the training data.

Deeper designs, however, may also be harder to train and may experience disappearing or bursting gradients. These problems may make it difficult for the model to pick up helpful information and result in subpar performance. Deeper architectures may also be more prone to overfitting, mainly if the model is overly complicated compared to the amount of training data.
Techniques like residual or skip connections can be used to reduce vanishing or exploding gradients, and regularization techniques can be used to avoid overfitting to address these problems.

6. When I examine the visualizations of the weights, the mlp1 and mlp2 architectures have shown a different weight visualization, and the CNN architectures have shown another weight visualization. The main difference between these two architecture types is the Max-Pooling layer since the size of the matrix decreases because of the Max-Pooling layer, and the output is not interpretable. On the other hand, the weight visualizations of the mlp1 and mlp2 architectures have shown patterns. If I examine these visualizations, the architectures tend to learn the curved figures and some straight lines. Especially in the visualizations of the mlp2 architecture, the curves are more precise, and the learning of the pattern can be seen easily. Regarding the CNN architectures, it is hard to comment on the weight visualizations since the outputs are just 3x3 matrices with white, gray, and black boxes. I tried to comment on these visualizations, but the patterns that the CNN architectures learned are not easily interpretable.

7. The mlp1 and mlp2 architectures are specialized to detect the classes that have more curved features, namely automobiles, cats, and dogs. It can be concluded from their weight visualizations that they have extracted more curved patterns, and the edges of the objects can be seen clearly. When it comes to CNN architectures, these models are more generalized architectures, and they are better at finding the patterns of the objects in the given input dataset. The hard part is to interpret the weight visualizations.

8. The weights of the mlp1 and mlp2 architectures are more interpretable since their curved and straight-line patterns are easily seen from their weight visualizations. On the other hand, CNN architectures are hard to comment on.

9. The multilayer perceptron models, mlp1 and mlp2, have completely connected layers. However, mlp2 has a deeper design than mlp1 because it has a second hidden layer. Both models have a linear output layer without a bias term and ReLU activation functions between layers.

In terms of completely connected layers and ReLU activation functions, mlp2 is similar to mlp1, but it has a more intricate architecture and an additional hidden layer. One hidden layer makes for the simpler architecture known as mlp1.

Three convolutional layers are followed by three fully connected layers in the convolutional neural network known as the cnn_3. In order to extract features from the input image, each convolutional layer applies a set of learnable filters. These features are then passed via non-linear activation functions, such as ReLU (Rectified Linear Unit), in order to incorporate non-linearity into the model. Following flattening, the output of the last convolutional layer is passed into the fully connected layers, which carry out the classification process.

Similar to the cnn_3, the cnn_4 has a fourth convolutional layer, making it a four-layer CNN. The additional convolutional layer enables the network to learn more intricate information from the input image, potentially enhancing its accuracy in classifying images.

In terms of their structures, cnn3, cnn4, and cnn5 are all similar in that they use convolutional layers to extract features from the input image, followed by fully connected layers to perform classification. However, cnn4 and cnn5 are more complex than cnn3, with additional convolutional layers that allow them to learn more complex features. Additionally, cnn5 has an additional max-pooling layer that helps it to learn translation-invariant features.

Similar to the cnn_4, the cnn_5 is a five-layer CNN, but it adds a convolutional layer before a max-pooling layer. The max-pooling layer shrinks the input image's spatial size and aids in the network's learning of translation-invariant features, which can increase the network's resistance to changes in the input image.

In terms of their structural similarities, CNNs cnn_3, cnn_4, and cnn_5 all use convolutional layers to extract features from the input image before performing classification using fully connected layers. In contrast, CNNs cnn_4 and cnn_5 are more sophisticated than cnn_3 and can learn more intricate characteristics since they have more convolutional layers. A further max-pooling layer on cnn_5 aids in the learning of translation-invariant features.

The performance of CNN architectures is superior to MLP architectures. One of the main reasons for this is that extracting features and recognizing patterns are more successful than MLP architectures thanks to the convolutional layers and max-pool layers in the architecture. In addition to these, CNN architectures also have performance differences among themselves. Although the cnn_3 architecture is a model with high generalization performance, its training and validation accuracy values are not as high as other CNN architectures. This shows that cnn_3 can have a better training score with more training. As for the cnn_4 architecture, although it has higher training accuracy than cnn_3, it did not perform well enough in terms of validation accuracy. Apart from these, the cnn_5 architecture showed a superior result than all other architectures in terms of both training accuracy and validation accuracy. This shows that CNN architectures are better at classification than MLP architectures. Among the CNN architectures, the most successful model is cnn_5.

10. I would choose the cnn_5 architecture for the classification task. There are many reasons for this, but first of all, I conclude that the cnn_5 architecture, which has a high training accuracy value, learns the data well. In addition, seeing that the validation accuracy value does not fall behind the training accuracy value shows that the generalization performance of the model is quite good, and it will give good results in different data types. In addition to these, the training loss value of the model is less than all other models, and thanks to this model, which has a very high test accuracy value, I think that I can achieve a good performance even with data that was not included in the training before.

Part4. Experimenting Activation Functions

1. The mlp1_sigmoid neural network has a smaller training loss than the mlp1_relu neural network, as can be seen by looking at the training losses. This shows that, at least in terms of reducing the loss function, the mlp1_sigmoid network is able to learn the training data better than the mlp1_relu network.

Next, we can see that the mlp1_relu neural network has greater gradient magnitudes than the mlp1_sigmoid network by comparing the gradient amplitudes. This is not unexpected given that, especially when the input values are big, the ReLU activation function can generate gradients that are more aggressive than those produced by the sigmoid function. The mlp1_sigmoid network, on the other hand, seems to have a more consistent gradient behavior, with lower magnitudes that only marginally rise over training. This could help prevent problems like gradient explosion or disappearance and improve generalization performance on untested data.

In conclusion, the choice of activation function may significantly affect a neural network's training loss and gradient behavior. When it comes to lowering the loss and avoiding gradient problems, sigmoid may do better than ReLU, which can result in greater gradients and possibly quicker learning.

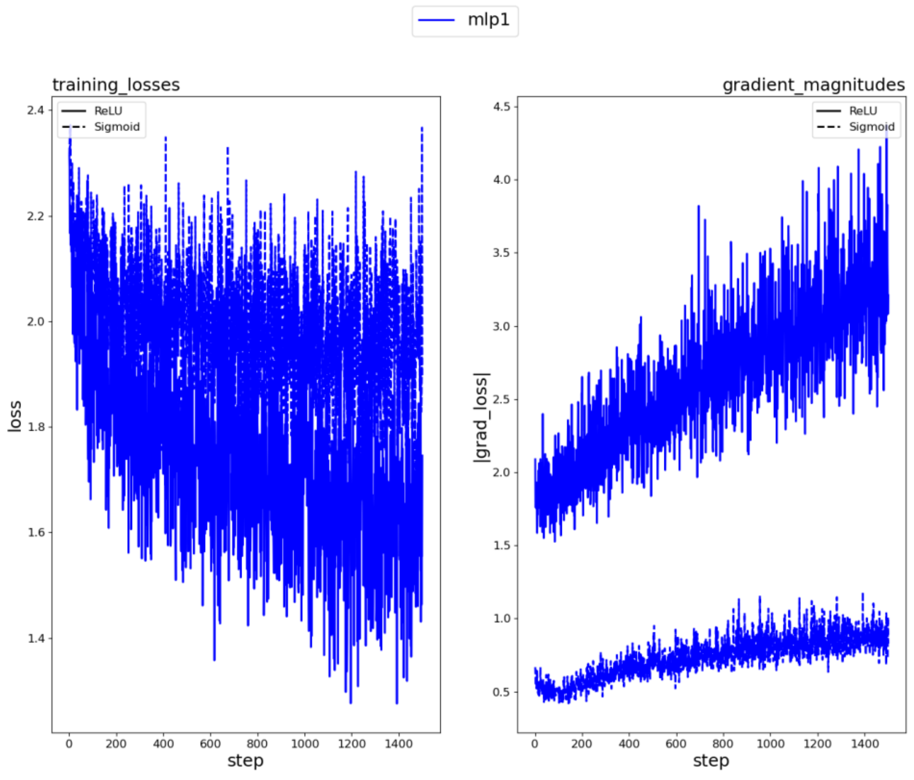

Figure 17. Training Losses and Gradient Magnitudes of Model mlp1

The mlp2_sigmoid network once more has a lower training loss than the mlp2_relu network, as can be seen by looking at the training losses. This shows that, at least in terms of reducing the loss function, the mlp2_sigmoid network is able to learn the training data better than the mlp2_relu network. As opposed to the previous example, the difference in training losses between the two networks is greater here, pointing to a stronger benefit for sigmoid activation.

The mlp2_relu neural network, when compared to the mlp2_sigmoid network, once more has bigger gradient magnitudes, as can be seen by looking at the gradient magnitudes. In contrast to the prior example, the magnitude difference between the two networks is smaller here, suggesting that the ReLU activation function may not be as aggressive in generating gradients. Although the magnitudes of the gradients are smaller and only marginally larger throughout training, the mlp2_sigmoid network nevertheless displays a more consistent gradient behavior.

To conclude, similar to the previous experiment, the results from this one also reveal that the mlp2_sigmoid network performs better in terms of minimizing training loss and displaying a more stable gradient behavior.

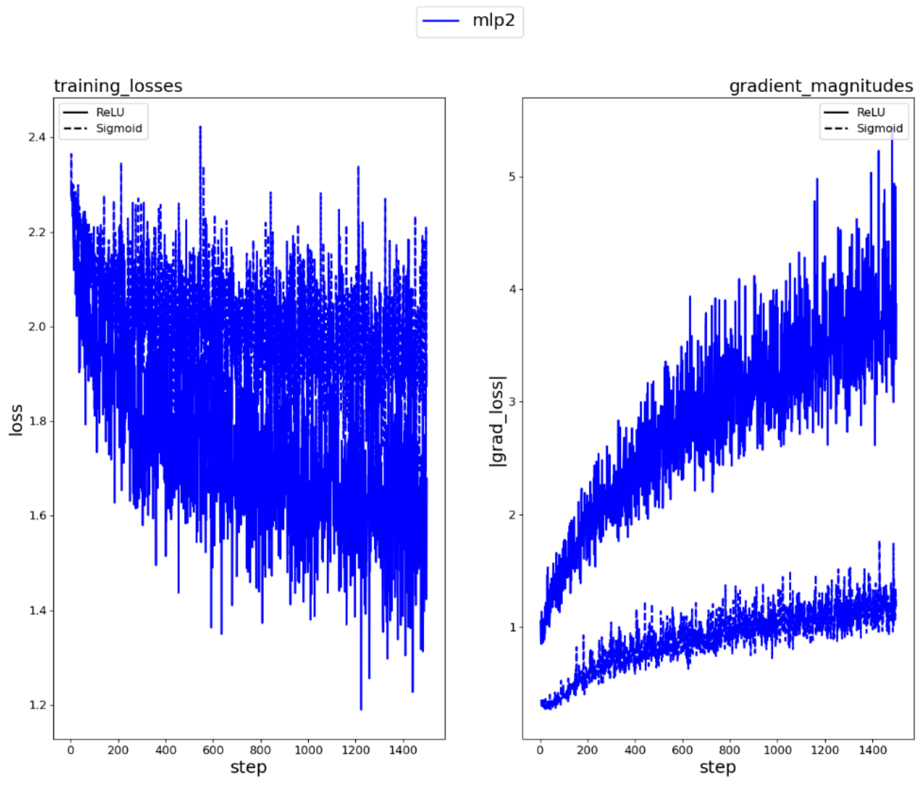

Figure 18. Training Losses and Gradient Magnitudes of Model mlp2

Next, we can see that the cnn_3_relu neural network has bigger gradient magnitudes than the cnn_3_sigmoid network by comparing the gradient amplitudes. Although the gradient magnitudes in the cnn_3_sigmoid network are quite tiny, this may indicate that the network is not picking up new information as quickly as the cnn_3_relu network. The relatively small gradients in the cnn_3_Sigmoid network might be slowing down learning, preventing the network from developing as quickly as it could.

In conclusion, this experiment's findings indicate that the cnn_3_sigmoid network performs better than the cnn_3_relu network in minimizing training loss. The cnn_3_sigmoid network's extremely modest gradient magnitudes, however, would suggest that the network is not learning as well as it could.

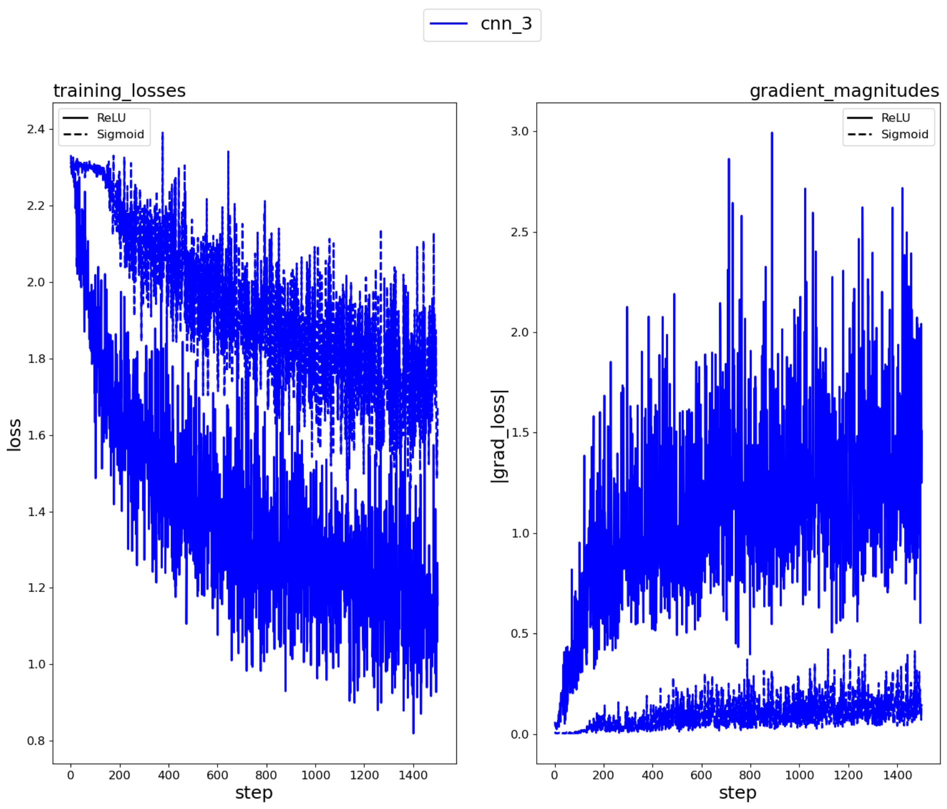

Figure 19. Training Losses and Gradient Magnitudes of Model cnn_3

The cnn_4_relu network has a substantially smaller training loss than the cnn_4_sigmoid network, as can be seen by looking at the training losses. In fact, the cnn_4_sigmoid network's training loss stayed constant during training, which raises the possibility that the network is not learning at all.

Next, we can see that the cnn_4_relu neural network has greater gradient magnitudes compared to the cnn_4_sigmoid network by examining the gradient magnitudes. The cnn_4_sigmoid network's gradient magnitudes are 0, which raises the possibility that the network's parameters aren't changing at all while being trained.

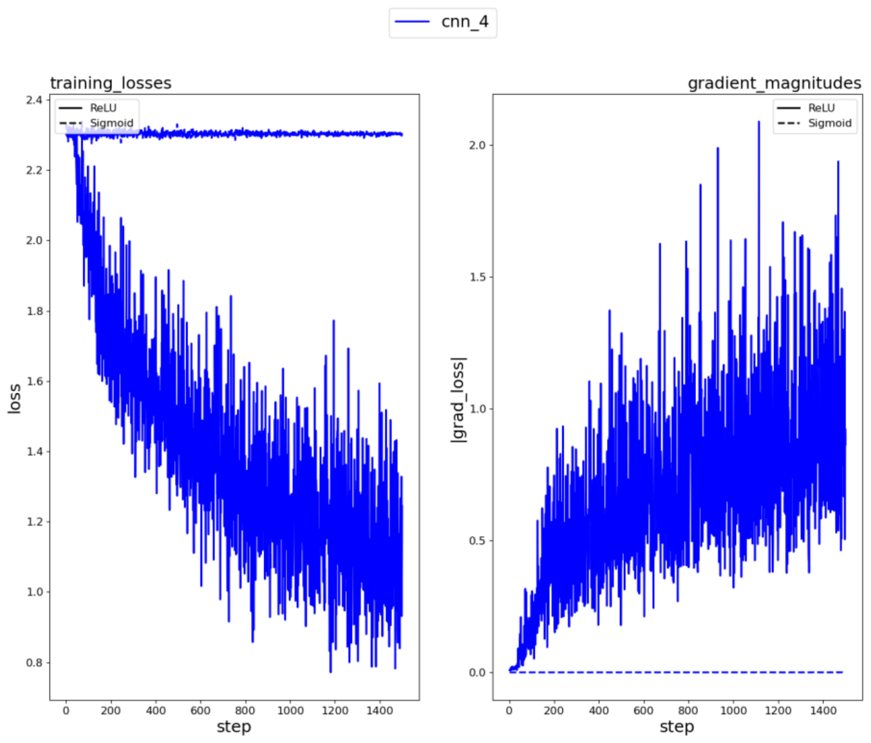

Figure 20. Training Losses and Gradient Magnitudes of Model cnn_4

When we examine the training performance of the cnn_5 architecture, it can be obtained that the gradient magnitude of the cnn_5_sigmoid is again zero and the training loss of the cnn_5_sigmoid remained unchanged. It means that the cnn_5_sigmoid architecture is not learning the data.

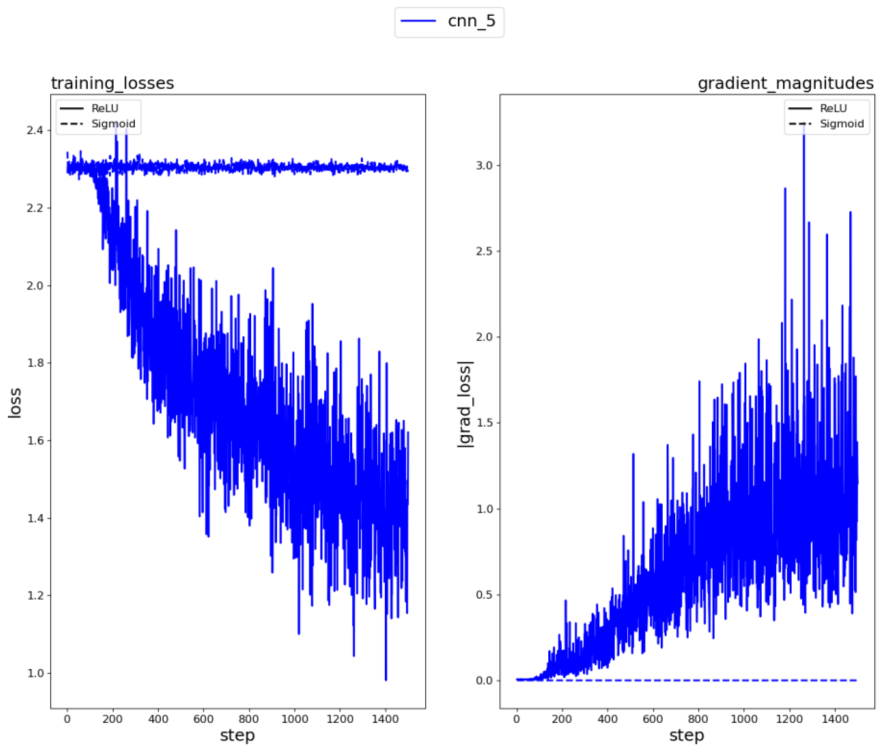

Figure 21. Training Losses and Gradient Magnitudes of Model cnn_5

The vanishing gradient problem is a frequent problem in deep neural networks that can happen when the gradient is very small (or zero) during backpropagation, which has the effect of giving the weights in the network's initial layers very little training data updates. The network's overall performance may suffer as a result of these early layers finding it challenging to acquire useful representations.

The vanishing gradient problem and neural network depth are closely related concepts. The likelihood of running across a vanishing gradient problem rises as a neural network's depth increases. This is because each layer might help to reduce the gradient's amplitude since the gradient is often back propagated via several layers.

In case of cnn_4_relu and cnn_5_relu architectures may have vanishing gradient problems as a result of the rather deep nature of these networks. Although the vanishing gradient issue is generally somewhat reduced by the ReLU activation function employed in these networks, it is still possible for the gradients to become very small in the deeper layers. The cnn_4_sigmoid and cnn_5_sigmoid networks' zero-gradient magnitudes indicate that the sigmoid activation function may be aggravating the vanishing gradient issue in these networks.

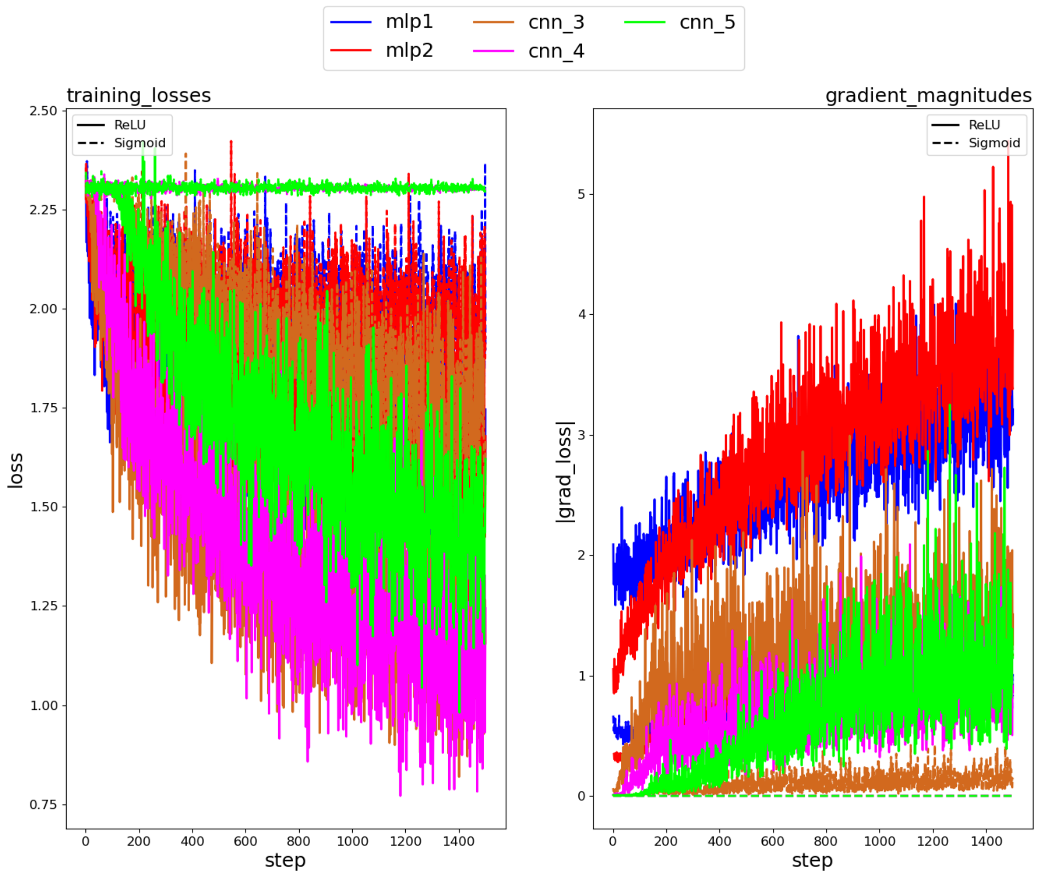

Figure 22. Training Losses and Gradient Magnitudes of All Models

2. The activation functions used in the network are one of several potential causes of the vanishing gradient problem. The gradients in the backpropagation process can become quite small as they are multiplied by the derivatives of activation functions like sigmoid and hyperbolic tangent since their derivatives are relatively small (between 0 and 1). As a result, as the gradients are back propagated through the layers of the network, they may "vanish" or approach zero, which may make it challenging for the earlier layers to acquire useful representations.

The depth of the network is another aspect that may play a role in the vanishing gradient issue. The issue of small gradients can be made worse as the network depth increases since more layers must be passed through in order to backpropagate the gradients. It's important to remember that deep networks can also experience the reverse issue, known as the expanding gradient problem. The network can diverge during training when the gradients are very steep, which causes the weights in the network to be changed by very large values.

In conclusion, the activation functions' small derivatives and the network's depth together might result in very small gradients as they are backpropagated through the layers, leading to the vanishing gradient problem.

3. Gradient scale: Compared to inputs in the [0.0, 1.0] range, inputs in the [0, 255] range can cause backpropagation gradients to grow significantly. Due of the relatively huge gradients that can result in excessively large weight updates and divergence, there may be problems with numerical stability during training.

Initialization of weights: To avoid saturating the activation functions, weights in neural networks are often initialized with low values. If the input values are in the [0, 255] range, the initial weights must be substantially less to account for the higher input values. If we utilize inputs in the range [0, 255], the network may take longer to converge, or it may not converge at all. This is due to the gradients' potential for instability, which can cause them to bounce between very large positive and negative values and prevent the weights from settling into an ideal arrangement.

Before feeding inputs into the neural network, they must first be normalized to the range [0.0, 1.0] if they are in the [0, 255] range. This involves an additional stage of preprocessing and may be computationally expensive.

Süleyman Yasin Peker
2305191

Part5. Experimenting Learning Rate

1. A hyperparameter called learning rate controls how frequently the model's parameters are changed during training. A higher learning rate often results in faster convergence since larger updates to the parameters can cover more ground in fewer rounds. However, an excessive learning rate can cause the updates to exceed the ideal values, which would eventually create instability and slower convergence.

The convergence speed, on the other hand, can be slowed down by a reduced learning rate because smaller updates to the parameters require more iterations to reach the optimal values. A slower learning rate, meanwhile, can also help the model avoid overshooting and instability.

While using the scheduled learning rate technique, the convergence speed of the model is increased up to 5 times because I also trained the same model with another optimizer called Adam and I obtained almost the same results as it was in the scheduled learning rate method which shows that the learning rate affects the convergence speed of the model. When it is too high, model can easily converge since high learning rate results in high convergence speed.

2. The deep learning model's ability to converge to a better point can be significantly influenced by the learning rate. A faster convergence to a better solution is facilitated by a higher learning rate, which enables the model to make larger modifications to the parameters. However, an excessive learning rate can cause the model to exceed the ideal solution, which will eventually create instability and slower convergence.

A reduced learning rate, on the other hand, results in smaller updates to the parameters, which can delay the convergence to a better solution. On the other hand, a slower learning rate can eventually aid the model in avoiding overshooting and converge to a more reliable and ideal result.

In order to ensure that the model converges to a better point, the choice of learning rate is crucial. Using strategies like learning rate schedules or adaptive learning rate methods like Adam or RMSprop, it is frequently utilized to start with a greater learning rate and gradually decrease it over time. With this method, the model is able to make larger updates early in the training process and refine the parameters as it gets closer to a better point of convergence.

3. The scheduled learning rate worked well while training the models. The main advantage of using the scheduled learning rate is to obtain high accuracy rates while training the model with smaller epoch. In other words, using scheduled learning rate is an efficient way to train the model. In deep learning the scheduled learning rate technique is used to modify the learning rate while training. The learning rate is a hyperparameter that controls how frequently a neural network's weights are updated. While a low learning rate can make the training process take longer, a high learning rate can cause the model to exceed the ideal weights. The learning rate is changed periodically throughout training by using the scheduled learning rate in order to improve performance.

The ability to avoid the model from becoming stuck in regional minima or plateaus is one benefit of adopting a planned learning rate. The model can explore various areas of the loss surface and discover a better global minimum by reducing the learning rate as training advances. Additionally, by enabling the model to reach the ideal weights more quickly, planned learning rate might hasten the training process. This is due to the fact that first large weight updates are made using a high learning rate, which is then gradually decreased for more precise weight adjustments. A significant method for enhancing the effectiveness and performance of machine learning models is planned learning rate.

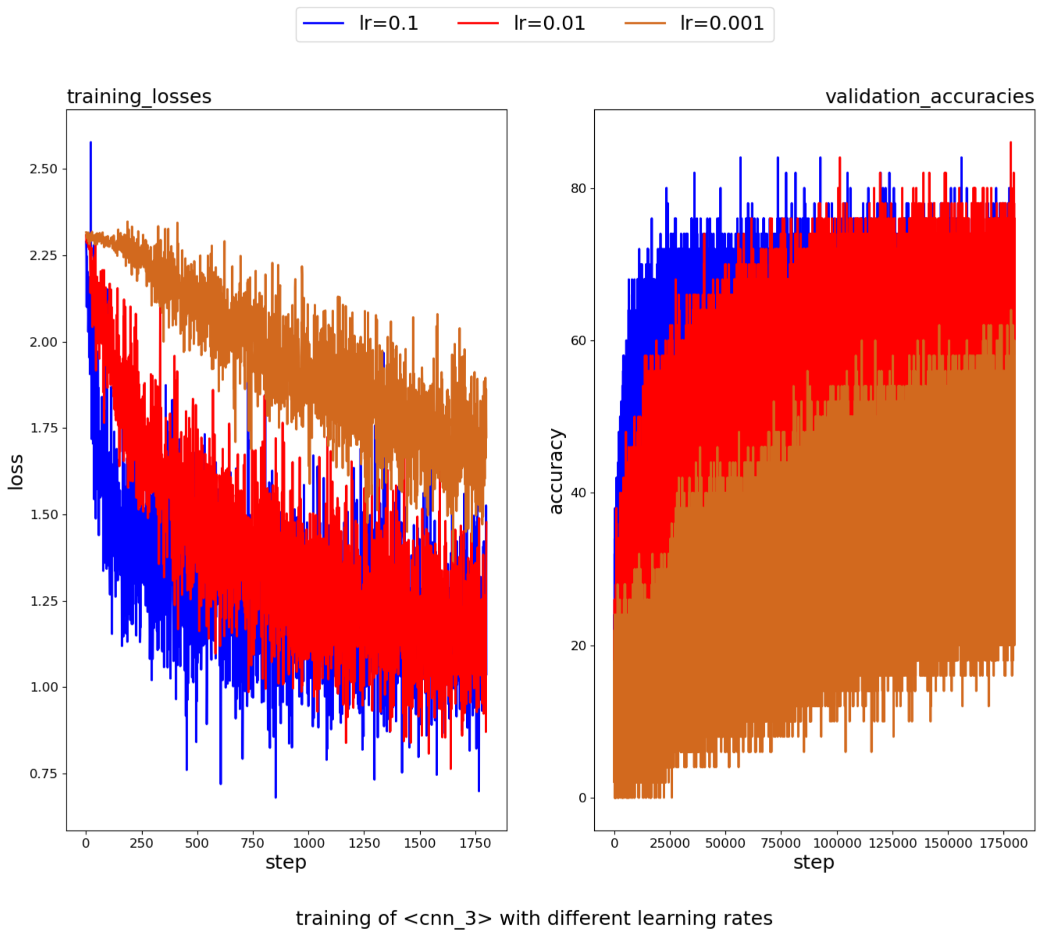

Figure 23. Training Loss and Validation Accuracy Plots of Model with Initial Learning Rates of 0.1, 0.01 and 0.001

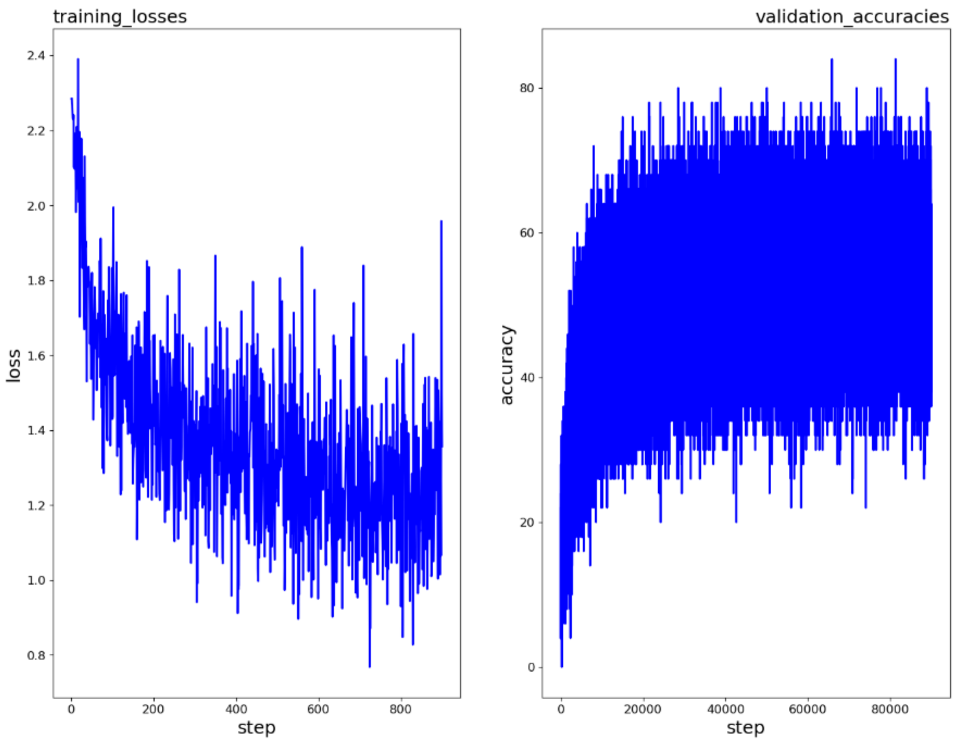

Figure 24. Training Loss and Validation Accuracy Plots of Model with Initial Learning Rate of 0.1 and 10 Epochs

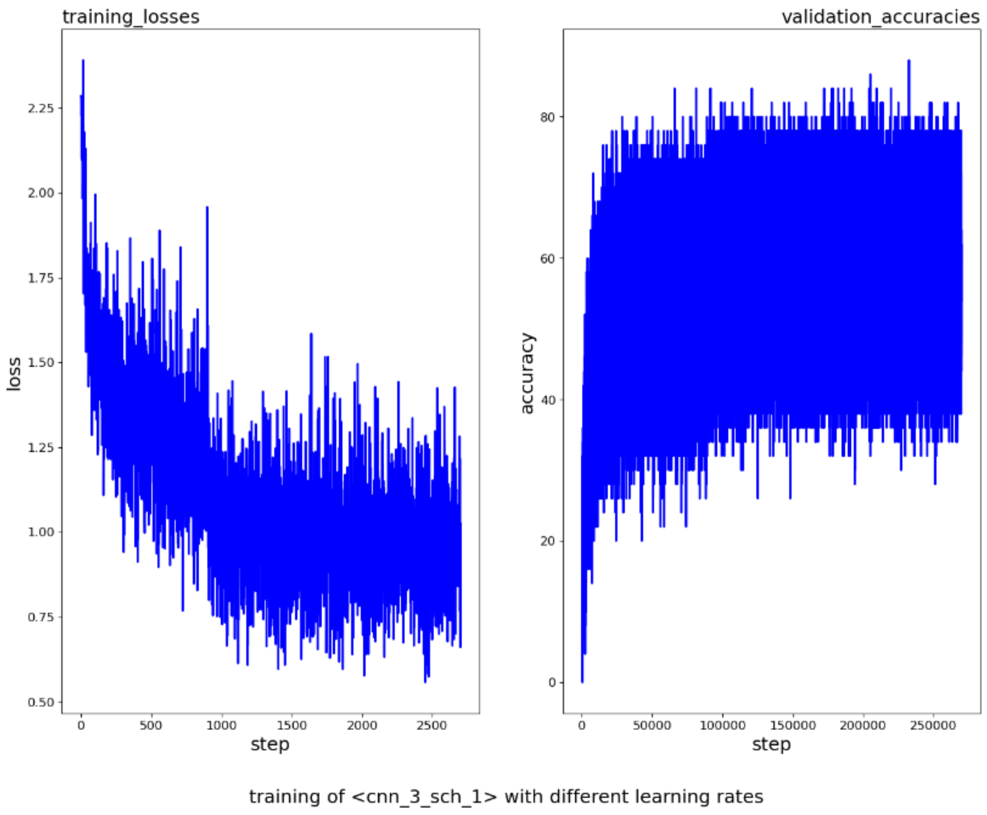

Figure 25. Training Loss and Validation Accuracy Plots of Model with Initial Learning Rate of 0.1 and After 10 Epochs, Learning Rate of 0.01 Until 30 Epochs

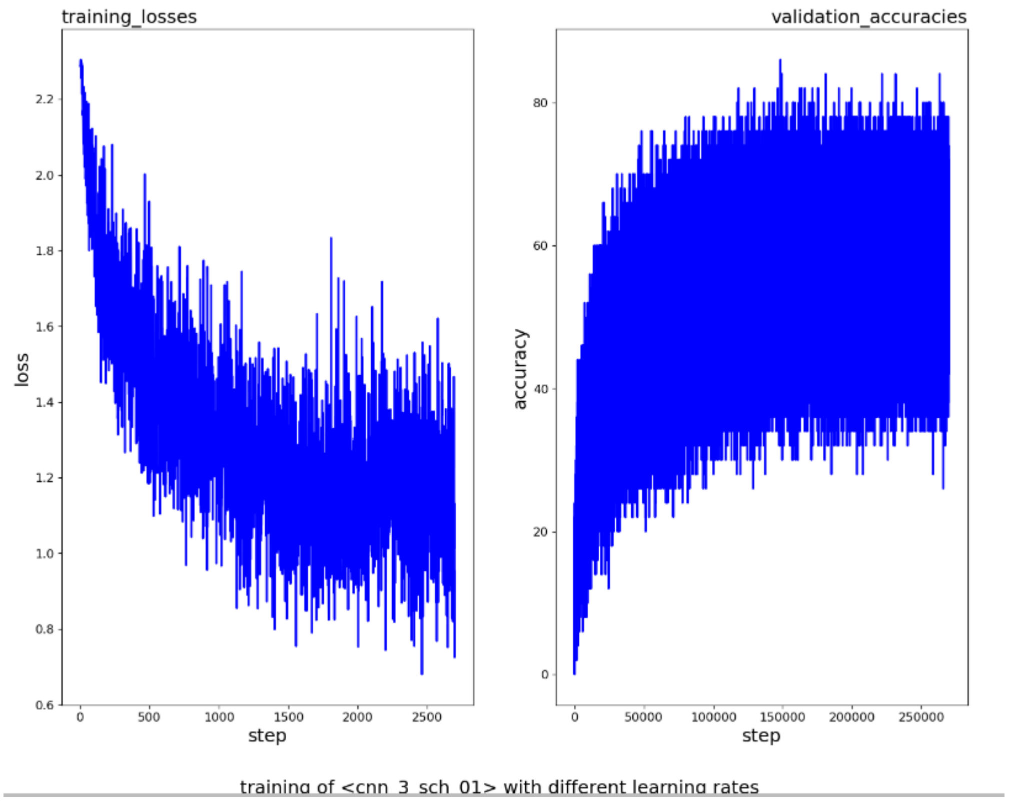

Figure 26. Training Loss and Validation Accuracy Plots of Model with Initial Learning Rate of 0.01 and After 15 Epochs, Learning Rate of 0.001 Until 30 Epochs

4. Comparison of the Test Results

In part 3, the cnn_3 model is trained 15 epochs and 20 runs. Then, the best test accuracy of the model is obtained as 58.92. Other test accuracies of the model were initially saved when the part 3 was being executed but in part 3, it is asked for us to create a pickle file and store the followings:

- 'name': string, indicating the user-defined name of the training.
- 'loss_curve': list of floats, indicating the loss at each step
- 'train_acc_curve': list of floats, indicating the training accuracy at each step
- 'val_acc_curve': list of floats indicating the validation accuracy at each step
- 'test_acc': float, indicating the **best test accuracy**
- 'weights': 2-D float array, weights of the first hidden layer of the trained MLP

In none of these above parameters, the test accuracy of all test procedure is saved. Only, the best test accuracy of the model was saved which was clearly stated in part 3. That's why, only the best test accuracy of the model cnn_3 and the best test accuracy of the model cnn_3_sch_01 is compared. Here are the best test accuracy results of these two models:

Best test accuracy of the cnn_3 model: **58.92**
Best test accuracy of the model cnn_3_sch_01: **58.26**

The total difference between the cnn_3 and cnn_3_sch_01 model is about 0.66 which is a negligibly small amount of difference. Especially, when we consider the training duration of these two models, it is obvious that the cnn_3 model was trained almost 7.5 hours. On the other hand, the cnn_3_sch_01 model was trained almost 1.5 hour which is one-fifth of the cnn_3 model. So, the test accuracies of these two models are almost the same but the convergence performance of the model cnn_3_sch_01 is superior the cnn_3 model. It can be concluded that when we use SGD optimizer and use scheduled learning rate to improve performance of the SGD based training, the model will be more efficient than the previous model which was using the Adam optimizer. In other words, to improve the SGD based learning and to reach high accuracies, scheduled learning rate should be applied to the models.

Süleyman Yasin Peker
2305191

Part2 Codes

```
import torch
import torch.nn
import torchvision
from torchvision import transforms
import numpy as np
from matplotlib import pyplot as plt
from torchvision.utils import make_grid
import os
def my_conv2d(input, kernel):
    # batch size => N
    # number of channels => C_in
    # height of the input planes in pixels => H_in
    # width of the input planes in pixels => W_in

    # input size: [N, C_in, H_in, W_in]
    # initialize the input parameters
    N = input.shape[0]
    C_in = input.shape[1]
    H_in = input.shape[2]
    W_in = input.shape[3]

    # kernel size: [C_out, C_in, H_kernel, W_kernel]
    # initialize the kernel parameters
    # initialize the kernel sizes
    C_out = kernel.shape[0]
    C_in = kernel.shape[1]
    H_kernel = kernel.shape[2]
    W_kernel = kernel.shape[3]

    # Calculate height of the output plane in pixels, H_out:
    H_out = H_in - H_kernel + 1

    # Calculate width of the output plane in pixels, W_out:
    W_out = W_in - W_kernel + 1

    # The output tensor should be initialized by zeros with these sizes:
    # output size: [N, C_out, H_out, W_out]

    output = np.zeros((N, C_out, H_out, W_out))

    # Convolution Operation can be described as:

    # output(N_i, C_outj) = bias(C_outj) + Sum[from k=0 to C_in-
1](weight(C_outj, k) * input(N_i, k)
    # where * is 2D Cross-correlation operator
    # This convolution operation can be implemented using nested for
loops. The outer for loop should start with the
    # batch size since we perform this operation for each batch. Then,
the next for loop should iterate through the
    # output channels since for each batch, we must calculate this
operation of all of the channels.
    # After that, the next for loop should iterate through the input
channels since the sum operation is from 0 to
    # C_in - 1. This describes the sum operation properly. Moving on, we
must calculate the convolution operation
```

```
    # from the first index of the input to the last index of the input,
which can be done by another two nested for
    # loops that iterates for output height and output width,
respectively.

    # In our case, there will be no bias, that's why the convolution
operation will be as follows:
    # output(N_i, C_outj) = Sum[from k=0 to C_in-1](weight(C_outj, k) *
input(N_i, k)

    # Convolution Operation
    for batch in range(N):
        for ch_out in range(C_out):
            for ch_in in range(C_in):
                for height in range(H_out):
                    for width in range(W_out):
                        output[batch, ch_out, height, width] +=
np.sum(input[batch, ch_in, height:height+H_kernel, width:width+W_kernel]
* kernel[ch_out, ch_in, :, :])
    return output
# Add part2Plots function
# Default settings of the torch.nn.Conv2d function:
# torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
#                    padding=0, dilation=1, groups=1, bias=True,
#                    padding_mode='zeros', device=None, dtype=None
# input shape: [batch size, input_channels, input_height, input_width]
input = np.load("samples_1.npy")
# kernel shape: [output_channels, input_channels, filter_height, filter
width]
kernel = np.load("kernel.npy")

out = my_conv2d(input, kernel)
# Plotting the result of the my_conv2d function
part2Plots(out)
```

Part3 Codes

```
import torch
import torch.nn
import torchvision
from torchvision import transforms
import numpy as np
from torch.utils.data.sampler import SubsetRandomSampler
from matplotlib import pyplot as plt
from torchvision.utils import make_grid
import pickle
import os
from matplotlib.lines import Line2D
# Define the used device
# Check whether cuda or cpu is used
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# If GPU is used, write cuda. Otherwise, CPU will be used for training
print(device)
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 
0.243, 0.261)),
    torchvision.transforms.Grayscale()
])

# Training set
train_data = torchvision.datasets.CIFAR10("./data", train = True, 
download = True, 
                                          transform = transform)
# Test set
test_data = torchvision.datasets.CIFAR10("./data", train = False, 
                                         transform = transform)
# Validation set that includes 10% of the training set
# So, firstly we should check the number of samples in the training set
train_length = len(train_data)

# Create a numpy array that stores the indices of train set from 0 to 
train_length
# In other words, [0, 50000)
train_indices = np.arange((train_length))

# Locate classes of each index so that they can be splitted equally
class_labels = np.array(train_data.targets)

# Take the 10% of the train set and store it as validation set length
validation_length = int(train_length*0.1)

# Number of samples per each class
class_sample_number = int(validation_length / len(train_data.classes))

# Create a list includes indices of each class
class_indices = [np.where(class_labels == i)[0] for i in 
range(len(train_data.classes))]

# Initialize the indices of the validation set
validation_indices = []
```

```
# Randomly chose indices per each class equally from the training set
for index in class_indices:
  validation_indices.extend(np.random.choice(index, class_sample_number,
replace = False))

# Calculate the train indices after the validation split
train_indices = list(set(train_indices) - set(validation_indices))

# Create a train sampler by excluding indices that are separated for the
validation set
train_sampler = SubsetRandomSampler(train_indices)

# Create a validation sampler by using the validation indices calculated
validation_sampler = SubsetRandomSampler(validation_indices)
# Define dataloaders to that are sampled accordingly
batch_size = 50
train_generator = torch.utils.data.DataLoader(train_data, batch_size =
batch_size, sampler = train_sampler)
test_generator = torch.utils.data.DataLoader(test_data, batch_size =
batch_size)
validation_generator =  torch.utils.data.DataLoader(train_data,
batch_size = batch_size, sampler = validation_sampler)
# Check whether the train, test and validation generators are created
correctly
# In the beginning, the sizes of the datasets were as follows:
# Train Set: 50000
# Test Set : 10000
# Validation Set: 0

# After splitting 10% of the training set into validation set, these
datasets are obtained:
# Train Set: 45000
# Test Set : 10000
# Validation Set: 5000

# According to the new values, the outputs of the DataLoaders should be
as follows:
# train_generator: 45000 / 50 = 900
# test_generator : 10000 / 50 = 200
# validation_generator: 5000 / 50 = 100
# Where first operand is the number of images per each dataset, and the
second operand is the batch size

# So, this assert tests whether the DataLoaders created correctly or not
assert( (len(train_generator) == 900) & (len(test_generator) == 200)  &
(len(validation_generator) == 100) )
# General definitions about the architectures:
# FC-N: Fully Connected layer of size N
# Conv-WxHxN: N Convolutional layers of size WxH
# MaxPool-2x2: Max-pooling operation of pool size 2x2
# PredictionLayer: FC-10

# Parameter Definitons:
# Stride: 1 for Convolutions
# Stride: 2 for Max-pooling operations
# Padding: Should be valid for both Convolution and Max-pooling
operations

# Optimizer: Adam (Adaptive Moment Estimation) with default setting
# Adam Optimizer with default settings:
# torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,
weight_decay=0,
#                   amsgrad=False, *, foreach=None, maximize=False,
capturable=False,
#                   differentiable=False, fused=None)

# Batch Size: 50 samples

# Three Dataset: Train Set, Test Set and Validation Set
# First Class: Multi Layer Perceptron 1 Class
# mlp1: [FC-32, ReLU] + PredictionLayer
class mlp1(torch.nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(mlp1, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size)
    self.fc2 = torch.nn.Linear(hidden_size, num_classes, bias = False)
    self.relu = torch.nn.ReLU()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden = self.fc1(x)
    relu = self.relu(hidden)
    output = self.fc2(relu)
    return output
# Second Class: Multi Layer Perceptron 2 Class
class mlp2(torch.nn.Module):
  def __init__(self, input_size, hidden_size1, hidden_size2,
num_classes):
    super(mlp2, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size1)
    self.fc2 = torch.nn.Linear(hidden_size1, hidden_size2, bias = False)
    self.fc3 = torch.nn.Linear(hidden_size2, num_classes, bias = False)
    self.relu = torch.nn.ReLU()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden1 = self.fc1(x)
    relu1 = self.relu(hidden1)
    hidden2 = self.fc2(relu1)
    relu2 = self.relu(hidden2)
    output = self.fc3(relu2)
    return output
# Third Class: Convolutional Neural Network 3
class cnn_3(torch.nn.Module):
  def __init__(self):
    super(cnn_3, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 'valid')
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
# Fourth Class: Convolutional Neural Network 4
class cnn_4(torch.nn.Module):
  def __init__(self):
    super(cnn_4, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu2 = torch.nn.ReLU()
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.relu3 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv4 = torch.nn.Conv2d(16, 16, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.relu4 = torch.nn.ReLU()
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc = torch.nn.Linear(16*4*4, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.maxpool1(x)
    x = self.conv4(x)
    x = self.relu4(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*4*4)
    x = self.fc(x)
    return x
# Fifth Class: Convolutional Neural Network 5
```

```
class cnn_5(torch.nn.Module):
  def __init__(self):
    super(cnn_5, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(8, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu2 = torch.nn.ReLU()
    self.conv3 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu3 = torch.nn.ReLU()
    self.conv4 = torch.nn.Conv2d(8, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu4 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv5 = torch.nn.Conv2d(16, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu5 = torch.nn.ReLU()
    self.conv6 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu6 = torch.nn.ReLU()
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc = torch.nn.Linear(16*4*4, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.maxpool1(x)
    x = self.conv4(x)
    x = self.relu4(x)
    x = self.conv5(x)
    x = self.relu5(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*4*4)
    x = self.fc(x)
    return x
# Change the name of the model as required
model = cnn_4()
model.to(device)
# If CNNs are used, type model.conv1. Otherwise, type model.fc1
params_cnn_4 = model.conv1.weight.data.clone().numpy()
# Traing the model
# Define the epoch number
epoch = 15

# Create train, validation and test accuracy lists that hold the accuracy
values
train_loss = []
train_accuracy = []
validation_accuracy = []
```

```
test_accuracy = []

# Create average train, validation and test accuracy lists that hold the
average of the accuracy values
avg_loss_curve = []
avg_train_acc_curve = []
avg_val_acc_curve = []

# Create variables that hold best model weights and best test accuracy
best_weights = None
best_test_acc = 0.0

# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)
total_run = 10
# Train and evaluate the model 10 times
for iteration in range(total_run):

  # Create loss: use cross entropy loss
  loss_func = torch.nn.CrossEntropyLoss()
  # Create the Adam optimizer with default parameters
  optimizer = torch.optim.Adam(model.parameters(), lr = 0.001, betas =
(0.9, 0.999), eps = 1e-08, weight_decay = 0)

  # For loop that iterates for each epoch
  for cur_epoch in range(epoch):
    # Transfer the model to train mode
    model.train()
    for batch_idx, (x_train, y_train) in enumerate(train_generator):
      # Transfer the input and the output to the used device (cpu or
cuda)
      x_train, y_train = x_train.to(device), y_train.to(device)

      # At every iteration reset the gradient to zero so that start from
scratch
      optimizer.zero_grad()

      # Make prediction by using the model
      y_prediction = model(x_train)

      # Calculate the loss
      loss = loss_func(y_prediction, y_train)

      # Backward pass and optimization step
      loss.backward()
      optimizer.step()

      if (batch_idx + 1) % 10 == 0:
        # Save training loss for every 10 steps
        train_loss.append(loss.item())

        # Transfer the model to eval mode
        model.eval()
        with torch.no_grad():

          # Calculate the training accuracy
          # Initialize the correct and total predictions
          correct_train = 0
          total_train = 0

          output = model(x_train)
          y_prediction = output.argmax(dim=1)

          for i in range(y_prediction.shape[0]):
            if y_train[i] == y_prediction[i]:
              correct_train += 1
            total_train += 1

          # Append the train accuracy result to the list
          train_acc = 100 * correct_train / total_train
          train_accuracy.append(train_acc)

        for x_validation, y_validation in validation_generator:
          x_validation = x_validation.to(device)
          y_validation = y_validation.to(device)

          # Save validation accuracy for every 10 steps
          model.eval()
          with torch.no_grad():

            # Calculate the validation accuracy
            # Initialize the correct and total predictions
            correct_validation = 0
            total_validation = 0

            output = model(x_validation)
            y_prediction = output.argmax(dim=1)

            for i in range(y_prediction.shape[0]):
              if y_validation[i] == y_prediction[i]:
                correct_validation += 1
              total_validation += 1

            # Append the validation accuracy result to the list
            validation_acc = 100 * correct_validation / total_validation
            validation_accuracy.append(validation_acc)

        print('Run [{}/{}], Epoch [{}/{}], Step [{}/{}], Training Acc:
{:.2f}%, Validation Acc: {:.2f}%'
                      .format(iteration+1, total_run, cur_epoch+1, epoch,
batch_idx + 1, len(train_generator), train_acc, validation_acc))
  # Calculate the test accuracy
  model.eval()
  with torch.no_grad():
    correct_test = 0
    total_test = 0

    for x_test, y_test in test_generator:
      x_test = x_test.to(device)
      y_test = y_test.to(device)

      output = model(x_test)
      y_prediction = output.argmax(dim=1)
      for i in range(y_prediction.shape[0]):
        if y_test[i] == y_prediction[i]:
          correct_test += 1
        total_test += 1

    # Append the result to the list
    test_accuracy.append(100 * correct_test / total_test)

    # Save the best test accuracy and best model weights
    for test_idx in range(len(test_accuracy)):
      if test_accuracy[test_idx] > best_test_acc:
        best_test_acc = test_accuracy[test_idx]
        best_weights = model.conv1.weight.data.clone().numpy()

  #best_test_acc.append(best_test_accucary)
  avg_loss_curve.append(np.mean(train_loss, axis = 0))
  avg_train_acc_curve.append(np.mean(train_accuracy, axis = 0))
  avg_val_acc_curve.append(np.mean(validation_accuracy, axis = 0))
# Check the difference between the initial weights and the best weights
(best_weights - params_cnn_4).sum()
train_result_dict = {
    'name': 'cnn_4',
    'loss_curve': avg_loss_curve,
    'train_acc_curve': avg_train_acc_curve,
    'val_acc_curve': avg_val_acc_curve,
    'test_acc': best_test_acc,
    'weights': best_weights
}

# Save the dictionary object to a file
filename = 'part3_cnn_4.pkl'
with open(filename, 'wb') as f:
    pickle.dump(train_result_dict, f)
# Include part3Plots and visualizeWeights functions to continue
results = [train_result_dict]

part3Plots(results, save_dir=r'C:/Users/Yasin', filename='part3Plots')
weights = train_result_dict['weights']
visualizeWeights(weights, save_dir='C:/Users/yasin',
filename='input_weights')
weights = params_cnn_4
visualizeWeights(weights, save_dir='C:/Users/yasin',
filename='before_train_weights')
```

Süleyman Yasin Peker
2305191

Part4 Codes

```
import torch
import torch.nn
import torchvision
from torchvision import transforms
import numpy as np
from torch.utils.data.sampler import SubsetRandomSampler
from matplotlib import pyplot as plt
from torchvision.utils import make_grid
import pickle
import os
from matplotlib.lines import Line2D
# Define the used device
# Check whether cuda or cpu is used
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# If GPU is used, write cuda. Otherwise, CPU will be used for training
print(device)
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247,
0.243, 0.261)),
    torchvision.transforms.Grayscale()
])

# Training set
train_data = torchvision.datasets.CIFAR10("./data", train = True,
download = True,
                                          transform = transform)
# Define dataloaders to that are sampled accordingly
batch_size = 50
train_generator = torch.utils.data.DataLoader(train_data, batch_size =
batch_size, shuffle = True)
# First Class: Multi Layer Perceptron 1 Class
# mlp1: [FC-32, ReLU] + PredictionLayer
class mlp1(torch.nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(mlp1, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size)
    self.fc2 = torch.nn.Linear(hidden_size, num_classes, bias = False)
    self.relu = torch.nn.ReLU()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden = self.fc1(x)
    relu = self.relu(hidden)
    output = self.fc2(relu)
    return output
# First Class: Multi Layer Perceptron 1 Class
# mlp1: [FC-32, ReLU] + PredictionLayer
class mlp1_sigmoid(torch.nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(mlp1_sigmoid, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size)
    self.fc2 = torch.nn.Linear(hidden_size, num_classes, bias = False)
    self.sigmoid = torch.nn.Sigmoid()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden = self.fc1(x)
    sigmoid = self.sigmoid(hidden)
    output = self.fc2(sigmoid)
    return output
# Second Class: Multi Layer Perceptron 2 Class
class mlp2(torch.nn.Module):
  def __init__(self, input_size, hidden_size1, hidden_size2,
num_classes):
    super(mlp2, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size1)
    self.fc2 = torch.nn.Linear(hidden_size1, hidden_size2, bias = False)
    self.fc3 = torch.nn.Linear(hidden_size2, num_classes, bias = False)
    self.relu = torch.nn.ReLU()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden1 = self.fc1(x)
    relu = self.relu(hidden1)
    hidden2 = self.fc2(relu)
    output = self.fc3(hidden2)
    return output
# Second Class: Multi Layer Perceptron 2 Class
class mlp2_sigmoid(torch.nn.Module):
  def __init__(self, input_size, hidden_size1, hidden_size2,
num_classes):
    super(mlp2_sigmoid, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size1)
    self.fc2 = torch.nn.Linear(hidden_size1, hidden_size2, bias = False)
    self.fc3 = torch.nn.Linear(hidden_size2, num_classes, bias = False)
    self.sigmoid = torch.nn.Sigmoid()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden1 = self.fc1(x)
    sigmoid = self.sigmoid(hidden1)
    hidden2 = self.fc2(sigmoid)
    output = self.fc3(hidden2)
    return output
# Third Class: Convolutional Neural Network 3
class cnn_3(torch.nn.Module):
  def __init__(self):
    super(cnn_3, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 'valid')
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
```

```
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
# Third Class: Convolutional Neural Network 3
class cnn_3_sigmoid(torch.nn.Module):
  def __init__(self):
    super(cnn_3_sigmoid, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid1 = torch.nn.Sigmoid()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.sigmoid2 = torch.nn.Sigmoid()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 'valid')
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.sigmoid1(x)
    x = self.conv2(x)
    x = self.sigmoid2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
# Fourth Class: Convolutional Neural Network 4
class cnn_4(torch.nn.Module):
  def __init__(self):
    super(cnn_4, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.relu2 = torch.nn.ReLU()
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.relu3 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv4 = torch.nn.Conv2d(16, 16, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.relu4 = torch.nn.ReLU()
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc = torch.nn.Linear(16*4*4, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.maxpool1(x)
    x = self.conv4(x)
    x = self.relu4(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*4*4)
    x = self.fc(x)
    return x
# Fourth Class: Convolutional Neural Network 4
class cnn_4_sigmoid(torch.nn.Module):
  def __init__(self):
    super(cnn_4_sigmoid, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid1 = torch.nn.Sigmoid()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid2 = torch.nn.Sigmoid()
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.sigmoid3 = torch.nn.Sigmoid()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv4 = torch.nn.Conv2d(16, 16, kernel_size = (5,5), stride = 1,
padding = 'valid')
    self.sigmoid4 = torch.nn.Sigmoid()
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc = torch.nn.Linear(16*4*4, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.sigmoid1(x)
    x = self.conv2(x)
    x = self.sigmoid2(x)
    x = self.conv3(x)
    x = self.sigmoid3(x)
    x = self.maxpool1(x)
    x = self.conv4(x)
    x = self.sigmoid4(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*4*4)
    x = self.fc(x)
    return x
# Fifth Class: Convolutional Neural Network 5
class cnn_5(torch.nn.Module):
  def __init__(self):
    super(cnn_5, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 8, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(8, 16, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.relu2 = torch.nn.ReLU()
    self.conv3 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.relu3 = torch.nn.ReLU()
    self.conv4 = torch.nn.Conv2d(8, 16, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.relu4 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2, 
padding = 0)
    self.conv5 = torch.nn.Conv2d(16, 16, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.relu5 = torch.nn.ReLU()
    self.conv6 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.relu6 = torch.nn.ReLU()
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2, 
padding = 0)
    self.fc = torch.nn.Linear(16*4*4, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.maxpool1(x)
    x = self.conv4(x)
    x = self.relu4(x)
    x = self.conv5(x)
    x = self.relu5(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*4*4)
    x = self.fc(x)
    return x
# Fifth Class: Convolutional Neural Network 5
class cnn_5_sigmoid(torch.nn.Module):
  def __init__(self):
    super(cnn_5_sigmoid, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 8, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.sigmoid1 = torch.nn.Sigmoid()
    self.conv2 = torch.nn.Conv2d(8, 16, kernel_size = (3,3), stride = 1, 
padding = 'valid')
    self.sigmoid2 = torch.nn.Sigmoid()
    self.conv3 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid3 = torch.nn.Sigmoid()
    self.conv4 = torch.nn.Conv2d(8, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid4 = torch.nn.Sigmoid()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv5 = torch.nn.Conv2d(16, 16, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid5 = torch.nn.Sigmoid()
    self.conv6 = torch.nn.Conv2d(16, 8, kernel_size = (3,3), stride = 1,
padding = 'valid')
    self.sigmoid6 = torch.nn.Sigmoid()
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc = torch.nn.Linear(16*4*4, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.sigmoid1(x)
    x = self.conv2(x)
    x = self.sigmoid2(x)
    x = self.conv3(x)
    x = self.sigmoid3(x)
    x = self.maxpool1(x)
    x = self.conv4(x)
    x = self.sigmoid4(x)
    x = self.conv5(x)
    x = self.sigmoid5(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*4*4)
    x = self.fc(x)
    return x
# Change the name of the model as required
model = cnn_5()
model.to(device)
# Change the name of the model as required
model_sigmoid = cnn_5_sigmoid()
model_sigmoid.to(device)
# Compare ReLU and Logistic Sigmoid Function
# Use SGD for training method
# Learning Rate: 0.01
# Momentum: 0.0 (no momentum)
# Batch Size: 50
# No weight regularization
# Traing the model
# Define the epoch number
epoch = 15

# Create train, validation and test accuracy lists that hold the accuracy
values
train_loss = []
gradient_loss_magnitudes = []

# Calculate the total number of batches from division of train set and
batch size
```

```
batch_number = int(len(train_generator.dataset)/batch_size)

# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()

# Create the Adam optimizer with default parametersBaseException
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum =
0.0)

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)

    # At every iteration reset the gradient to zero so that start from
scratch
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)

    # Backward pass and optimization step
    loss.backward()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss and Loss Gradient for every 10 steps
      train_loss.append(loss.item())

      # Magnitudes of the Loss Gradients of the First Layer
      # Change XXX part according to the first layer of the architecture:
model.XXX.weight.grad.nump()
      loss_gradient_first_layer = model.conv1.weight.grad.norm().item()
      gradient_loss_magnitudes.append(loss_gradient_first_layer)

      # Save the weights corresponding to the first layer
      first_layer_weights = model.conv1.weight.data.cpu().numpy()


      print('Epoch [{}/{}], Step [{}/{}], Train Loss = {:.2f}%, Loss
Gradient = {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(),
                            loss_gradient_first_layer))
    optimizer.step()
# Traing the model
# Define the epoch number
epoch = 15

# Create train, validation and test accuracy lists that hold the accuracy
values
train_loss_sigmoid = []
```

```
gradient_loss_magnitudes_sigmoid = []

# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)

# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()

# Create the Adam optimizer with default parametersBaseException
optimizer = torch.optim.SGD(model_sigmoid.parameters(), lr = 0.01,
momentum = 0.0)

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model_sigmoid.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)

    # At every iteration reset the gradient to zero so that start from
scratch
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model_sigmoid(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)

    # Backward pass and optimization step
    loss.backward()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss and Loss Gradient for every 10 steps
      train_loss_sigmoid.append(loss.item())

      # Magnitudes of the Loss Gradients of the First Layer
      # Change XXX part according to the first layer of the architecture:
model_sigmoid.XXX.weight.grad.nump()
      loss_gradient_first_layer_sigmoid =
model_sigmoid.conv1.weight.grad.norm().item()

gradient_loss_magnitudes_sigmoid.append(loss_gradient_first_layer_sigmoid
)

      # Save the weights corresponding to the first layer
      first_layer_weights = model_sigmoid.conv1.weight.data.cpu().numpy()


      print('Epoch [{}/{}], Step [{}/{}], Train Loss = {:.2f}%, Loss
Gradient = {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(),
                            loss_gradient_first_layer_sigmoid))
    optimizer.step()
```

```
train_result_dict = {
    'name': 'cnn_5',
    'relu_loss_curve': train_loss,
    'sigmoid_loss_curve': train_loss_sigmoid,
    'relu_grad_curve': gradient_loss_magnitudes,
    'sigmoid_grad_curve': gradient_loss_magnitudes_sigmoid
}

# Save the dictionary object to a file
filename = 'part4_cnn_5.pkl'
with open(filename, 'wb') as f:
    pickle.dump(train_result_dict, f)
# To continue, add part4Plots function
results = [train_result_dict]

part4Plots(results, save_dir=r'C:/Users/Yasin', filename='part4Plots')
# To create the representations of all models, upload their pickle files
# open a file, where you stored the pickled data
file1 = open('part4_mlp1.pkl', 'rb')

# dump information to that file
data1 = pickle.load(file1)

# close the file
file1.close()

# open a file, where you stored the pickled data
file2 = open('part4_mlp2.pkl', 'rb')

# dump information to that file
data2 = pickle.load(file2)

# close the file
file2.close()

# open a file, where you stored the pickled data
file3 = open('part4_cnn_3.pkl', 'rb')

# dump information to that file
data3 = pickle.load(file3)

# close the file
file3.close()

# open a file, where you stored the pickled data
file4 = open('part4_cnn_4.pkl', 'rb')

# dump information to that file
data4 = pickle.load(file4)

# close the file
file4.close()

# open a file, where you stored the pickled data
file5 = open('part4_cnn_5.pkl', 'rb')

# dump information to that file
```

```
data5 = pickle.load(file5)

# close the file
file5.close()
results = [data1, data2, data3, data4, data5]

part4Plots(results, save_dir=r'C:/Users/Yasin', filename='part4Plots')
```

Part5 Codes

```
import torch
import torch.nn
import torchvision
from torchvision import transforms
import numpy as np
from torch.utils.data.sampler import SubsetRandomSampler
from matplotlib import pyplot as plt
from torchvision.utils import make_grid
import pickle
import os
from matplotlib.lines import Line2D
# Define the used device
# Check whether cuda or cpu is used
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# If GPU is used, write cuda. Otherwise, CPU will be used for training
print(device)
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247,
0.243, 0.261)),
    torchvision.transforms.Grayscale()
])

# Training set
train_data = torchvision.datasets.CIFAR10("./data", train = True,
download = True,
                                          transform = transform)
# Test set
test_data = torchvision.datasets.CIFAR10("./data", train = False,
                                         transform = transform)
# Validation set that includes 10% of the training set
# So, firstly we should check the number of samples in the training set
train_length = len(train_data)

# Create a numpy array that stores the indices of train set from 0 to
train_length
# In other words, [0, 50000)
train_indices = np.arange((train_length))

# Locate classes of each index so that they can be splitted equally
class_labels = np.array(train_data.targets)

# Take the 10% of the train set and store it as validation set length
validation_length = int(train_length*0.1)

# Number of samples per each class
class_sample_number = int(validation_length / len(train_data.classes))

# Create a list includes indices of each class
class_indices = [np.where(class_labels == i)[0] for i in
range(len(train_data.classes))]

# Initialize the indices of the validation set
validation_indices = []
```

```
# Randomly chose indices per each class equally from the training set
for index in class_indices:
  validation_indices.extend(np.random.choice(index, class_sample_number,
replace = False))

# Calculate the train indices after the validation split
train_indices = list(set(train_indices) - set(validation_indices))

# Create a train sampler by excluding indices that are separated for the
validation set
train_sampler = SubsetRandomSampler(train_indices)

# Create a validation sampler by using the validation indices calculated
validation_sampler = SubsetRandomSampler(validation_indices)
# Define dataloaders to that are sampled accordingly
batch_size = 50
train_generator = torch.utils.data.DataLoader(train_data, batch_size =
batch_size, sampler = train_sampler)
test_generator = torch.utils.data.DataLoader(test_data, batch_size =
batch_size)
validation_generator =  torch.utils.data.DataLoader(train_data,
batch_size = batch_size, sampler = validation_sampler)
# Check whether the train, test and validation generators are created
correctly
# In the beginning, the sizes of the datasets were as follows:
# Train Set: 50000
# Test Set : 10000
# Validation Set: 0

# After splitting 10% of the training set into validation set, these
datasets are obtained:
# Train Set: 45000
# Test Set : 10000
# Validation Set: 5000

# According to the new values, the outputs of the DataLoaders should be
as follows:
# train_generator: 45000 / 50 = 900
# test_generator : 10000 / 50 = 200
# validation_generator: 5000 / 50 = 100
# Where first operand is the number of images per each dataset, and the
second operand is the batch size

# So, this assert tests whether the DataLoaders created correctly or not
assert( (len(train_generator) == 900) & (len(test_generator) == 200)  &
(len(validation_generator) == 100) )
# Third Class: Convolutional Neural Network 3
class cnn_3_1(torch.nn.Module):
  def __init__(self):
    super(cnn_3_1, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 0)
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 0)
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 0)
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
# Third Class: Convolutional Neural Network 3
class cnn_3_01(torch.nn.Module):
  def __init__(self):
    super(cnn_3_01, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 0)
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 0)
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 0)
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
# Third Class: Convolutional Neural Network 3
class cnn_3_001(torch.nn.Module):
  def __init__(self):
    super(cnn_3_001, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 0)
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 0)
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 0)
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
model = cnn_3_1()
model.to(device)
model_01 = cnn_3_01()
model_01.to(device)
model_001 = cnn_3_001()
model_001.to(device)
# Traing the model
# Define the epoch number
epoch = 20

# Create train loss, validation and test accuracy lists
train_loss = []
validation_accuracy = []

# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)

# Train and evaluate the model
# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()
# Create the SGD optimizer with default parameters
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum = 0)

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)

    # At every iteration reset the gradient to zero so that start from
scratch
```

```
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)

    # Backward pass and optimization step
    loss.backward()
    optimizer.step()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss for every 10 steps
      train_loss.append(loss.item())

      # Save validation accuracy for every 10 steps
      for x_validation, y_validation in validation_generator:
        x_validation = x_validation.to(device)
        y_validation = y_validation.to(device)

        # Transfer the model to eval mode
        model.eval()
        with torch.no_grad():

          # Calculate the validation accuracy
          # Initialize the correct and total predictions
          correct_validation = 0
          total_validation = 0

          output = model(x_validation)
          y_prediction = output.argmax(dim=1)

          for i in range(y_prediction.shape[0]):
            if y_validation[i] == y_prediction[i]:
              correct_validation += 1
            total_validation += 1

          # Append the validation accuracy result to the list
          validation_acc = 100 * correct_validation / total_validation
          validation_accuracy.append(validation_acc)

      print('Epoch [{}/{}], Step [{}/{}], Training Loss: {:.2f}%,
Validation Acc: {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(), validation_acc))
# Traing the model
# Define the epoch number
epoch = 20

# Create train loss, validation and test accuracy lists
train_loss_01 = []
validation_accuracy_01 = []

# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)
```

```
# Train and evaluate the model
# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()
# Create the SGD optimizer with default parameters
optimizer = torch.optim.SGD(model_01.parameters(), lr = 0.01, momentum =
0)

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model_01.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)

    # At every iteration reset the gradient to zero so that start from
scratch
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model_01(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)

    # Backward pass and optimization step
    loss.backward()
    optimizer.step()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss for every 10 steps
      train_loss_01.append(loss.item())

      # Save validation accuracy for every 10 steps
      for x_validation, y_validation in validation_generator:
        x_validation = x_validation.to(device)
        y_validation = y_validation.to(device)

        # Transfer the model to eval mode
        model_01.eval()
        with torch.no_grad():

          # Calculate the validation accuracy
          # Initialize the correct and total predictions
          correct_validation = 0
          total_validation = 0

          output = model_01(x_validation)
          y_prediction = output.argmax(dim=1)

          for i in range(y_prediction.shape[0]):
            if y_validation[i] == y_prediction[i]:
              correct_validation += 1
            total_validation += 1

          # Append the validation accuracy result to the list
          validation_acc_01 = 100 * correct_validation / total_validation
          validation_accuracy_01.append(validation_acc_01)

      print('Epoch [{}/{}], Step [{}/{}], Training Loss: {:.2f}%,
Validation Acc: {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(), validation_acc_01))
# Traing the model
# Define the epoch number
epoch = 20

# Create train loss, validation and test accuracy lists
train_loss_001 = []
validation_accuracy_001 = []

# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)

# Train and evaluate the model
# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()
# Create the SGD optimizer with default parameters
optimizer = torch.optim.SGD(model_001.parameters(), lr = 0.001, momentum
= 0)

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model_001.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)

    # At every iteration reset the gradient to zero so that start from
scratch
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model_001(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)

    # Backward pass and optimization step
    loss.backward()
    optimizer.step()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss for every 10 steps
      train_loss_001.append(loss.item())

      # Save validation accuracy for every 10 steps
      for x_validation, y_validation in validation_generator:
        x_validation = x_validation.to(device)
        y_validation = y_validation.to(device)
```

```
         # Transfer the model to eval mode
         model_001.eval()
         with torch.no_grad():

           # Calculate the validation accuracy
           # Initialize the correct and total predictions
           correct_validation = 0
           total_validation = 0

           output = model_001(x_validation)
           y_prediction = output.argmax(dim=1)

           for i in range(y_prediction.shape[0]):
             if y_validation[i] == y_prediction[i]:
               correct_validation += 1
             total_validation += 1

           # Append the validation accuracy result to the list
           validation_acc_001 = 100 * correct_validation /
total_validation
           validation_accuracy_001.append(validation_acc_001)

      print('Epoch [{}/{}], Step [{}/{}], Training Loss: {:.2f}%,
Validation Acc: {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(), validation_acc_001))
# To continue, add part5Plots function
train_result_dict = {
    'name': 'cnn_3',
    'loss_curve_1': train_loss,
    'loss_curve_01': train_loss_01,
    'loss_curve_001': train_loss_001,
    'val_acc_curve_1': validation_accuracy,
    'val_acc_curve_01': validation_accuracy_01,
    'val_acc_curve_001': validation_accuracy_001
}

# Save the dictionary object to a file
filename = 'part5_cnn_3.pkl'
with open(filename, 'wb') as f:
    pickle.dump(train_result_dict, f)
results = [train_result_dict]

part5Plots(results, save_dir=r'C:/Users/Yasin', filename='part5Plots')
# CNN3 Architecture Used For Scheduled Learning Rate
class cnn_3_scheduled_1(torch.nn.Module):
  def __init__(self):
    super(cnn_3_scheduled_1, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 0)
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 0)
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
```

```
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 0)
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
model_sch_1 = cnn_3_scheduled_1()
model_sch_1.to(device)
# Create train loss and validation accuracy lists
train_loss_sch_1 = []
validation_accuracy_sch_1 = []
# Create the SGD optimizer
# Change the learning rate from 0.1 to 0.01 after 10 epochs and run the
training code again
optimizer = torch.optim.SGD(model_sch_1.parameters(), lr = 0.01, momentum
= 0)

# After running 10 epoch, change the epoch to 20
epoch = 20
# Traing the model
# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)

# Train and evaluate the model
# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model_sch_1.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)

    # At every iteration reset the gradient to zero so that start from
scratch
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model_sch_1(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)
    # Backward pass and optimization step
    loss.backward()
    optimizer.step()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss for every 10 steps
      train_loss_sch_1.append(loss.item())

      # Save validation accuracy for every 10 steps
      for x_validation, y_validation in validation_generator:
        x_validation = x_validation.to(device)
        y_validation = y_validation.to(device)

        # Transfer the model to eval mode
        model_sch_1.eval()
        with torch.no_grad():

          # Calculate the validation accuracy
          # Initialize the correct and total predictions
          correct_validation = 0
          total_validation = 0

          output = model_sch_1(x_validation)
          y_prediction = output.argmax(dim=1)

          for i in range(y_prediction.shape[0]):
            if y_validation[i] == y_prediction[i]:
              correct_validation += 1
            total_validation += 1

          # Append the validation accuracy result to the list
          validation_acc_sch_1 = 100 * correct_validation /
total_validation
          validation_accuracy_sch_1.append(validation_acc_sch_1)

      print('Epoch [{}/{}], Step [{}/{}], Training Loss: {:.2f},
Validation Acc: {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(), validation_acc_sch_1))
train_result_dict_sch_1 = {
    'name': 'cnn_3_sch_1',
    'loss_curve_1': train_loss_sch_1,
    'val_acc_curve_1': validation_accuracy_sch_1,
}

# Save the dictionary object to a file
filename = 'part5_cnn_3_sch_1.pkl'
with open(filename, 'wb') as f:
    pickle.dump(train_result_dict, f)
results_sch_1 = [train_result_dict_sch_1]

part5Plots(results_sch_1, save_dir=r'C:/Users/Yasin',
filename='part5Plots_sch_1')
# CNN3 Architecture Used For Scheduled Learning Rate
class cnn_3_scheduled_01(torch.nn.Module):
  def __init__(self):
    super(cnn_3_scheduled_01, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, kernel_size = (3,3), stride = 1,
padding = 0)
    self.relu1 = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, kernel_size = (5,5), stride = 1,
padding = 0)
    self.relu2 = torch.nn.ReLU()
    self.maxpool1 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.conv3 = torch.nn.Conv2d(8, 16, kernel_size = (7,7), stride = 1,
padding = 0)
    self.maxpool2 = torch.nn.MaxPool2d(kernel_size = (2,2), stride = 2,
padding = 0)
    self.fc  = torch.nn.Linear(16*3*3, 10, bias = False)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool1(x)
    x = self.conv3(x)
    x = self.maxpool2(x)
    x = x.view(-1, 16*3*3)

    x = self.fc(x)
    return x
model_sch_01 = cnn_3_scheduled_01()
model_sch_01.to(device)
# Create train loss and validation accuracy lists
train_loss_sch_01 = []
validation_accuracy_sch_01 = []
# Create the SGD optimizer
# Change the learning rate from 0.01 to 0.001 after 10 epochs and run the
training code again
optimizer = torch.optim.SGD(model_sch_01.parameters(), lr = 0.001,
momentum = 0)

# After running 15 epoch, remain the epoch number the same as
epoch = 15
# Traing the model
# Calculate the total number of batches from division of train set and
batch size
batch_number = int(len(train_generator.dataset)/batch_size)

# Train and evaluate the model
# Create loss: use cross entropy loss
loss_func = torch.nn.CrossEntropyLoss()

# For loop that iterates for each epoch
for cur_epoch in range(epoch):
  # Transfer the model to train mode
  model_sch_01.train()
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    # Transfer the input and the output to the used device (cpu or cuda)
    x_train, y_train = x_train.to(device), y_train.to(device)
    # At every iteration reset the gradient to zero so that start from
scratch
    optimizer.zero_grad()

    # Make prediction by using the model
    y_prediction = model_sch_01(x_train)

    # Calculate the loss
    loss = loss_func(y_prediction, y_train)

    # Backward pass and optimization step
    loss.backward()
    optimizer.step()

    if (batch_idx + 1) % 10 == 0:
      # Save training loss for every 10 steps
      train_loss_sch_01.append(loss.item())

      # Save validation accuracy for every 10 steps
      for x_validation, y_validation in validation_generator:
        x_validation = x_validation.to(device)
        y_validation = y_validation.to(device)

        # Transfer the model to eval mode
        model_sch_01.eval()
        with torch.no_grad():

          # Calculate the validation accuracy
          # Initialize the correct and total predictions
          correct_validation = 0
          total_validation = 0

          output = model_sch_01(x_validation)
          y_prediction = output.argmax(dim=1)

          for i in range(y_prediction.shape[0]):
            if y_validation[i] == y_prediction[i]:
              correct_validation += 1
            total_validation += 1

          # Append the validation accuracy result to the list
          validation_acc_sch_01 = 100 * correct_validation /
total_validation
          validation_accuracy_sch_01.append(validation_acc_sch_01)

      print('Epoch [{}/{}], Step [{}/{}], Training Loss: {:.2f},
Validation Acc: {:.2f}%'
                    .format(cur_epoch+1, epoch, batch_idx + 1,
len(train_generator), loss.item(), validation_acc_sch_01))
train_result_dict_sch_01 = {
    'name': 'cnn_3_sch_01',
    'loss_curve_1': train_loss_sch_01,
    'val_acc_curve_1': validation_accuracy_sch_01,
}

# Save the dictionary object to a file
filename = 'part5_cnn_3_sch_01.pkl'
```

```
with open(filename, 'wb') as f:
    pickle.dump(train_result_dict, f)
results_sch_01 = [train_result_dict_sch_01]

part5Plots(results_sch_01, save_dir=r'C:/Users/Yasin',
filename='part5Plots_sch_01')
# Calculate the test accuracy
test_accuracy_01 = []
best_test_acc_01 = 0.0
model_sch_01.eval()
with torch.no_grad():
  correct_test = 0
  total_test = 0

  for x_test, y_test in test_generator:
    x_test = x_test.to(device)
    y_test = y_test.to(device)

    output = model_sch_01(x_test)
    y_prediction = output.argmax(dim=1)

    for i in range(y_prediction.shape[0]):
      if y_test[i] == y_prediction[i]:
        correct_test += 1
      total_test += 1

  # Append the result to the list
  test_accuracy_01.append(100 * correct_test / total_test)

  # Save the best test accuracy and best model weights
  for test_idx in range(len(test_accuracy_01)):
    if test_accuracy_01[test_idx] > best_test_acc_01:
      best_test_acc_01 = test_accuracy_01[test_idx]
# Open the pickle file of the cnn_3 model in part 3
file = open('part3_cnn_3.pkl', 'rb')

# dump information to that file
data = pickle.load(file)

# close the file
file.close()
# Best test accuracy of the model trained in part3
data['test_acc']
# Best test accuracy of the model trained in the scheduled learning
best_test_acc_01
# Difference between two best test accuracies
best_test_acc_01 - data['test_acc']
```