EE449 Homework-1

Tunahan Aktaş

Table of Contents

1. Basic Concepts

1.1. Which Function?

1.1.1. What function does an ANNs classifier trained with cross-entropy loss approximates?

Assume we have an ANN having N neurons at the input layer and M neurons at the output layer. Then, after training with cross-entropy loss function, our network will classify the given N inputs as one the M classes at the output. Therefore, we will approximate a function such that $f: R^N \rightarrow R^M$.

1.1.2. How is the loss defined to approximate that function?

Cross entropy loss is defined as in equation 1, where t_j is the supervised label and y_j ANNs result of the j^{th} output neuron. The summation iterates over all output neurons.

$$L = -\sum_j t_j . \log{(y_j)} \tag{1}$$

1.1.3. Why?

Considering equation 1, only the neuron which represents labeled class will have $\mathrm{t_j} = 1$ and will affect the loss calculation.

$$L = -\log(y_i)$$

For the worst case, for a given input, if the label is t_i and our ANN gives zero probability as y_i, our loss will be:

$$L = -\lim_{y_i \rightarrow 0^+} (\log(y_i)) = \infty$$

And for the best case, $y_i = 1.0$:

$$L = -\log(y_i) = 0$$

Therefore, we are penalizing the difference between the labelled class and our prediction. As the difference increases, the penalty increases greatly.

1.2.Gradient Computation

$$\nabla_w L \mid_{w=w_k} = \frac{w_{k+1} - w_k}{\gamma} \tag{2}$$

1.3.Some Training Parameters and Basic Parameter Calculations

1.3.1. What are batch and epoch in the context of MLP training?

- **Epoch** is the number of training iterations.
- **Batch** is the number of training samples in each iteration.

In stochastic gradient descent, they are determined as hyperparameters before the training. We have predetermined batches in each of the epoch.

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

- N/B

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

- We have 1 iteration per batch in SGD. We already calculated batch size as N/B. Then, #iteration = E.(N/B)

1.4. Computing Number of Parameters of ANN Classifiers

1.4.1. Derive a formula to compute the number of parameters that the MLP has

- Considering the bias term and all input nodes, input layer has $D_{in} + 1$ connections to each neuron of the first hidden layer; therefore, we have $(D_{in} + 1).H_1$ parameters from input to first hidden layer.
- A neuron hidden layer j has $H_{j-1} + 1$ (considering bias) connections from the previous layer. Then, we have $(H_{j-1} + 1).H_j$ connections from layer $j - 1$ to j.

$$\#parameters = (D_{in} + 1).H_1 + \left(\sum_{j=2}^{K}(H_{j-1} + 1).H_j\right) + (H_K + 1).D_{out}$$

1.4.2. Derive a formula to compute the number of parameters that the CNN has

For each filter, we have $H_k.W_k$ parameters and 1 bias parameter. At each layer, we have C_i filters. Therefore, we can calculate the number of parameters in CNN case as:

$$\#parameters = \sum_{k=1}^{K-1}(H_k.W_k + 1).C_k$$

2. Experimenting ANN Architectures

NOTE: Since the curves are too noisy, I took the average of each consecutive 10 samples and create new curves for almost all graphs below. That is, if the curve is x dimensional vector, denoised curve is x/10 dimensional. An example:

Noisy curve: `dic1["val_acc_curve_1"]`

Denoised curve: `np.mean(dic1["val_acc_curve_1"].reshape(-1, 10), axis=1)`

2.1.Performance Comparison Plots

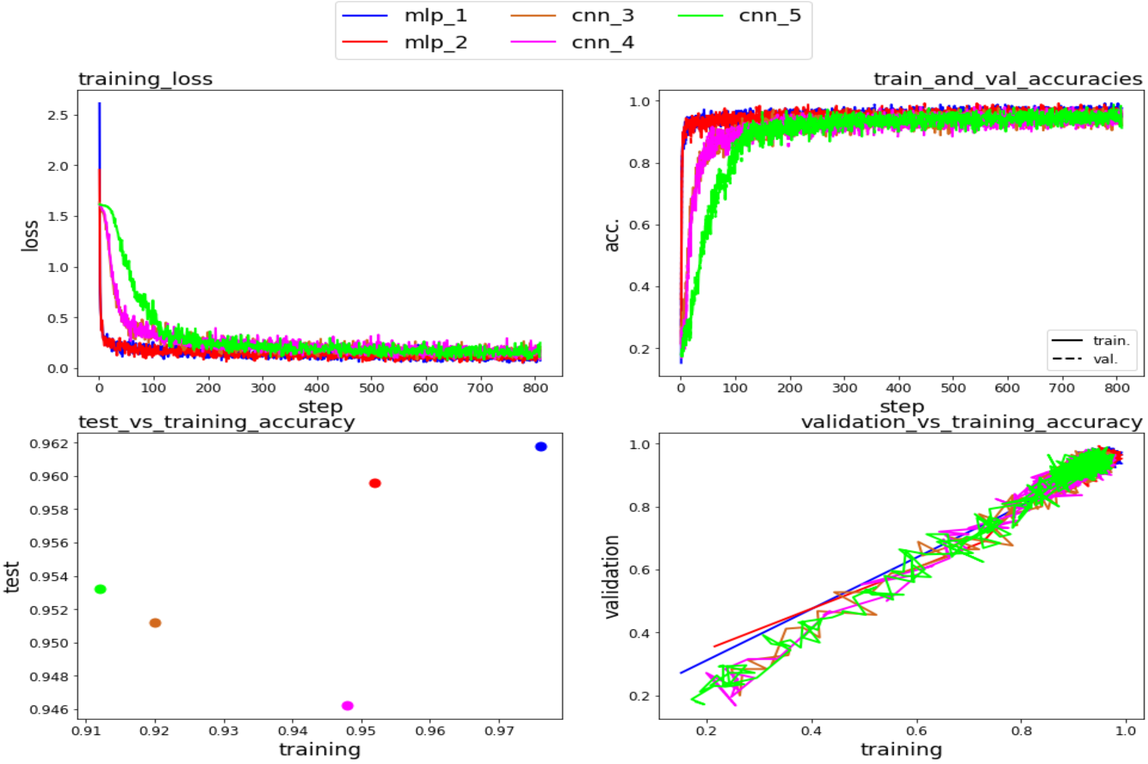

Figure 1: Performance comparison plots for Part 2.

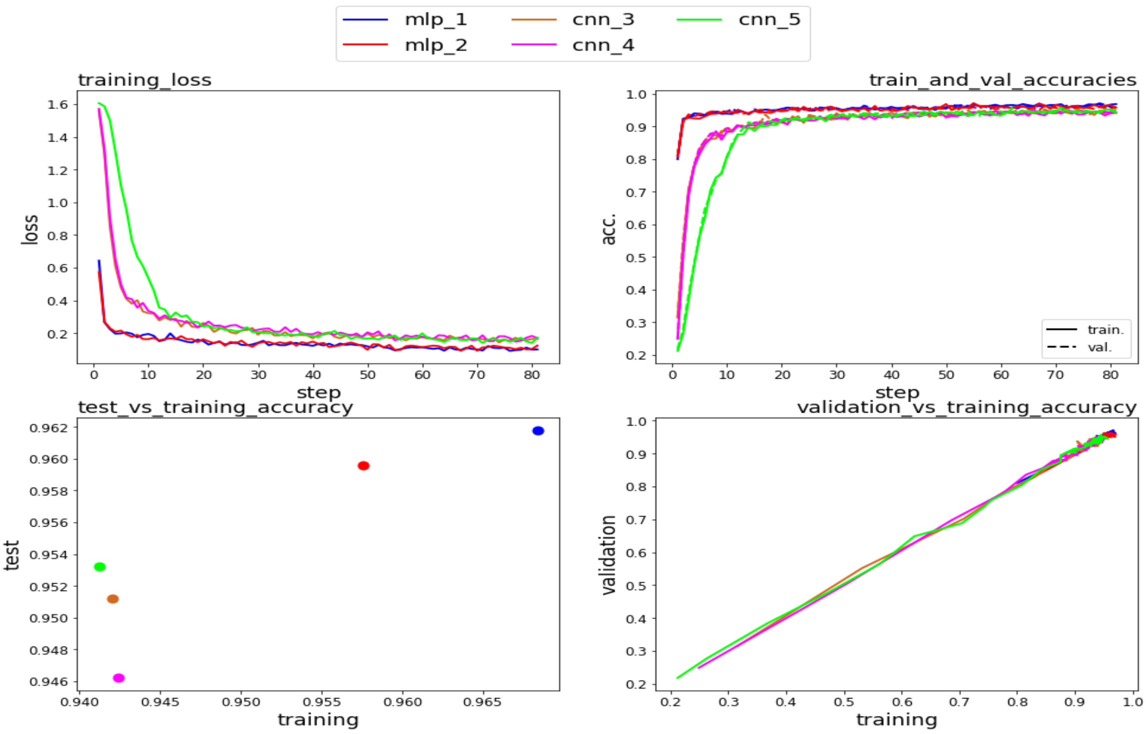

Figure 2: Denoised performance comparison plots of Figure 1

2.2.Weight Visualization

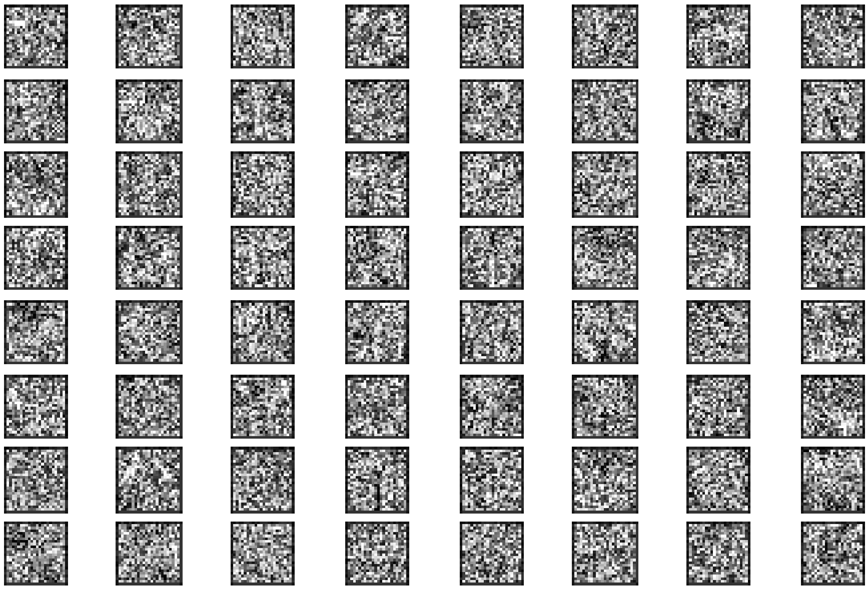

Figure 3: Visualized first layer weights of mlp_1.

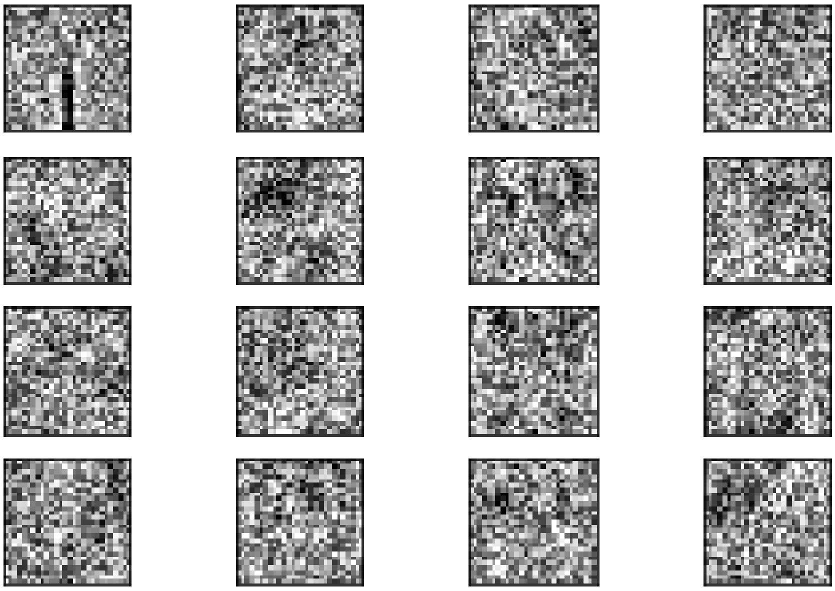

Figure 4: Visualized first layer weights of mlp_2.

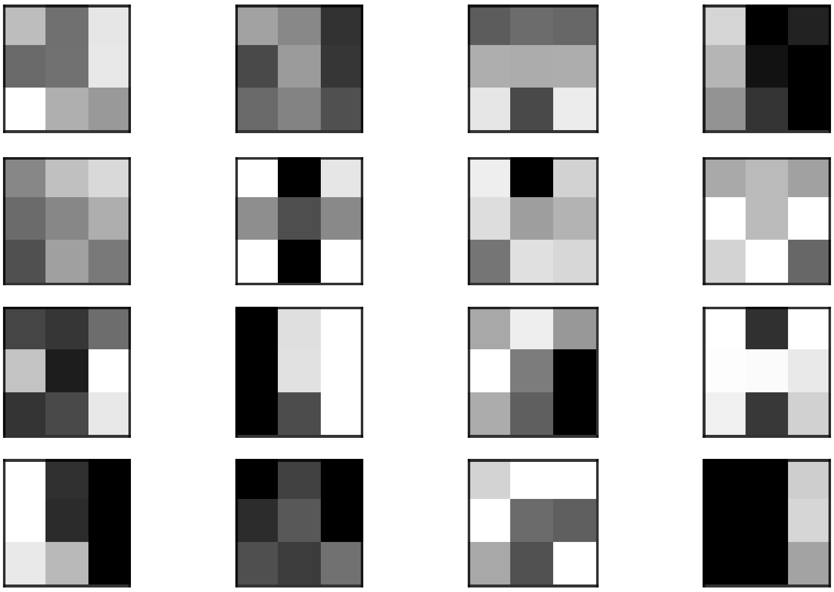

Figure 5: Visualized first layer weights of cnn_3.

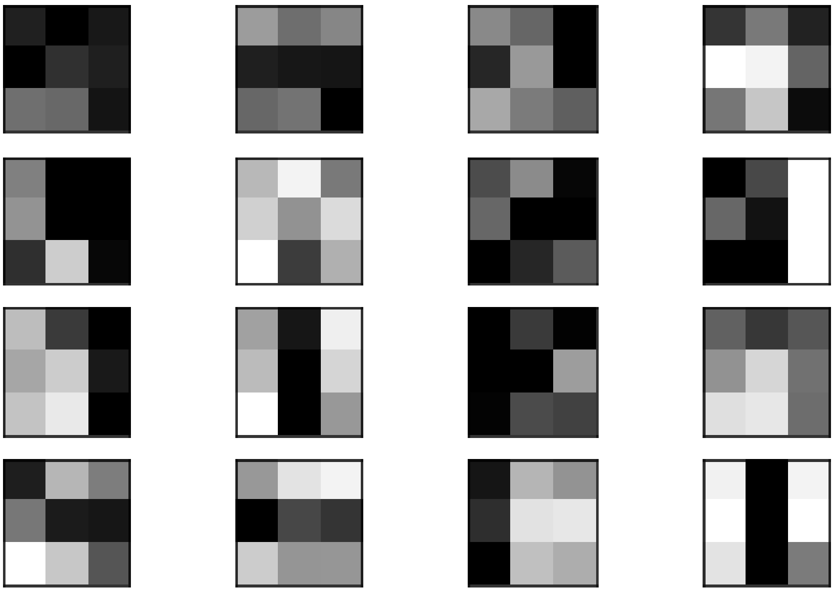

Figure 6: Visualized first layer weights of cnn_4.

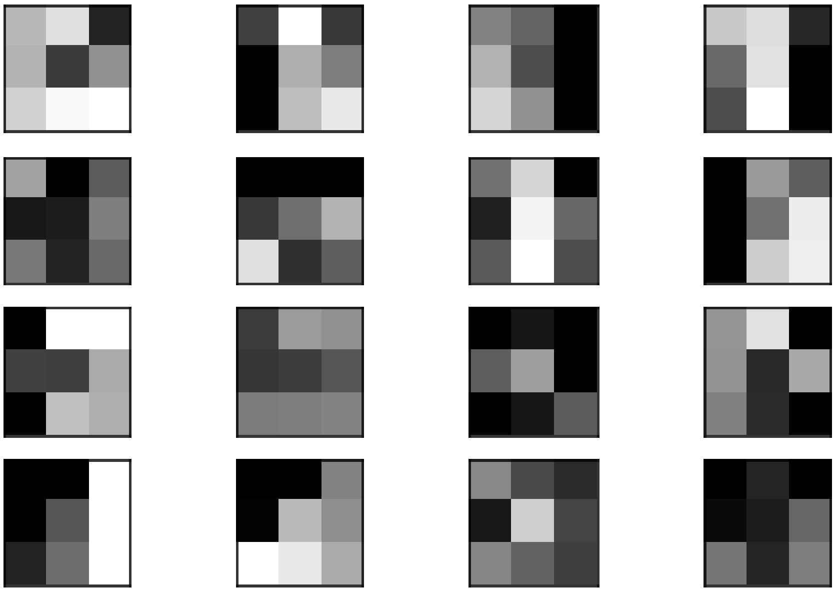

Figure 7: Visualized first layer weights of cnn_5.

2.3. Discussions

2.3.1. What is the Generalization Performance of a Classifier?

Generalization performance of a classifier is a measure of how well the model is at classifying images that it did not see during training phase.

2.3.2. Which Plots are Informative to Inspect Generalization Performance?

The plots must compare training phase accuracy with validation or testing phase accuracies. Therefore, the graphs that are named as *train_and_val_accuracies*, *validation_vs_training_accuracy* and *test_vs_training_accuracy* in Figure 1 are informative to inspect generalization performance.

2.3.3. Compare the Generalization Performance of the Architectures

If we look at the *test_vs_training_accuracy* in Figure 1, the difference between the training and test accuracy is smallest for cnn_5 case. Therefore, cnn_5 has the best generalization performance. In general, CNNs seems to have better generalization performance, compared to MLPs.

2.3.4. How Does the Number of Parameters Affect the Classification and Generalization Performance?

- Number of parameters: mlp_1>mlp_2>cnn_3>cnn_4>cnn_5
- The best generalization performance: cnn_5
- In general, generalization performance: CNNs>MLPs

Therefore, generalization performance and number of parameters are inversely proportional.

2.3.5. How Does the Depth of the Architecture Affect the Classification and Generalization Performance?

- Depth of the architectures: cnn_5> cnn_4>cnn_3 mlp_2> mlp_1
- Generalization performance: CNNs>MLPs

- Classification performance for MLPs: mlp_1>mlp_2.
- Classification performance for CNNs: cnn_5>cnn_4>cnn_3.

Therefore, the depth of the ANN is **directly proportional** to **generalization performance** and **inversely proportional** to **classification performance**.

2.3.6. Considering the Visualizations of the Weights, are They Interpretable?

Since the weights are in the first layer, they learn low level features. Therefore, it is hard to interpret from the visualizations. However, considering CNN weights, we can say that the filters try to detect sharp and smooth edges and corners.

2.3.7. Can You Say Whether the Units are Specialized to Specific Classes?

No, I cannot say that. Maybe higher-level units are specialized. First layer only detects edges, that is, black to white or white to black transitions.

2.3.8. Weights of Which Architecture are More Interpretable?

If we look at Figure 6, it seems cnn_4 is more interpretable because it includes vertical and horizontal edge filters, almost complete white and black filters. However, cnn_5 has also some intuitive filters, like corner filters.

2.3.9. Comment on the Structures

MLPs are akin to each other, and CNNs are akin to each other. If we compare convolutional layers; from cnn_3 to cnn_5, we are reducing the kernel size and increasing the filter number. Thus, we are reducing the number of parameters and increasing the depth. Since we have fewer parameters, memorization decreases and generalization increases. The accuracy is also increasing from cnn_3 to cnn_5.
MLP is the opposite. Mlp_2 architecture has lower parameters and higher depth but both generalization and accuracy is better in mlp_1 architecture.

2.3.10. Which Architecture Would You Pick for This Classification Task? Why?

Although the mlp_1 has the best accuracy, I would choose cnn_5 because it is the best in generalization and the accuracy difference is not so big.

3. Experimenting Activation Functions

3.1. Plots

3.1.1. Performance Comparison Plot for All Architectures

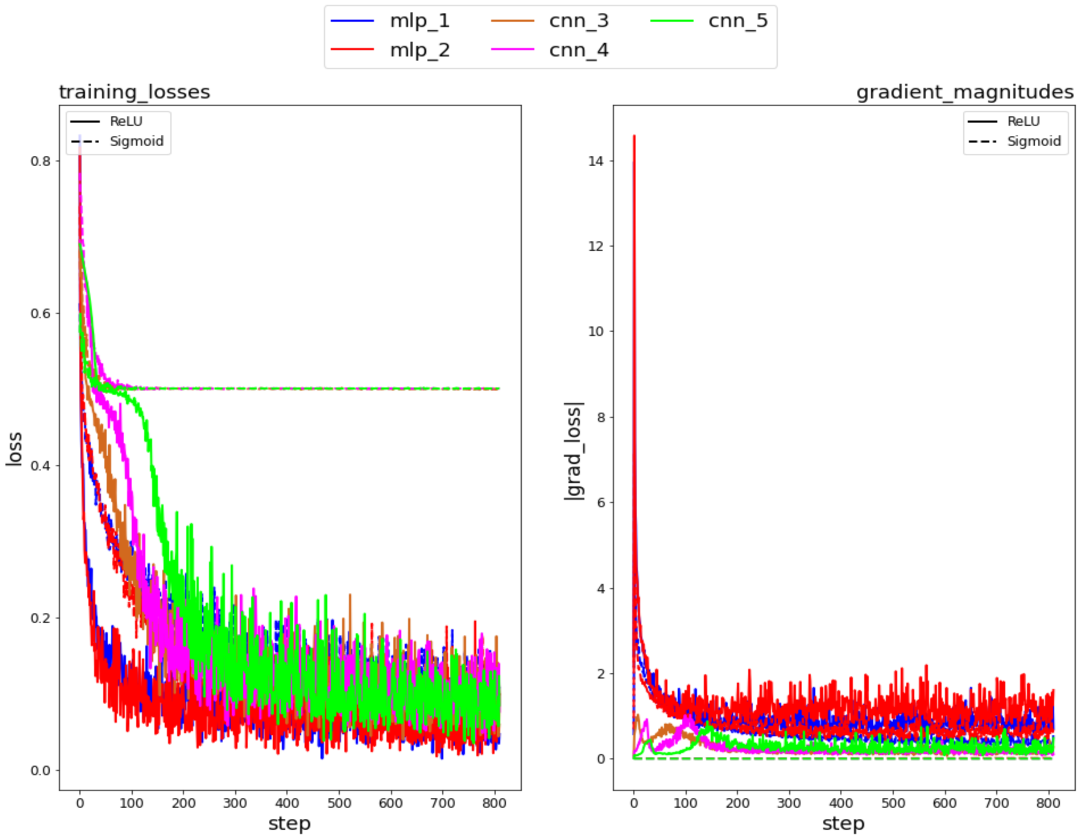

Figure 8: Performance comparison plot. Compares the results of all the ANNs.

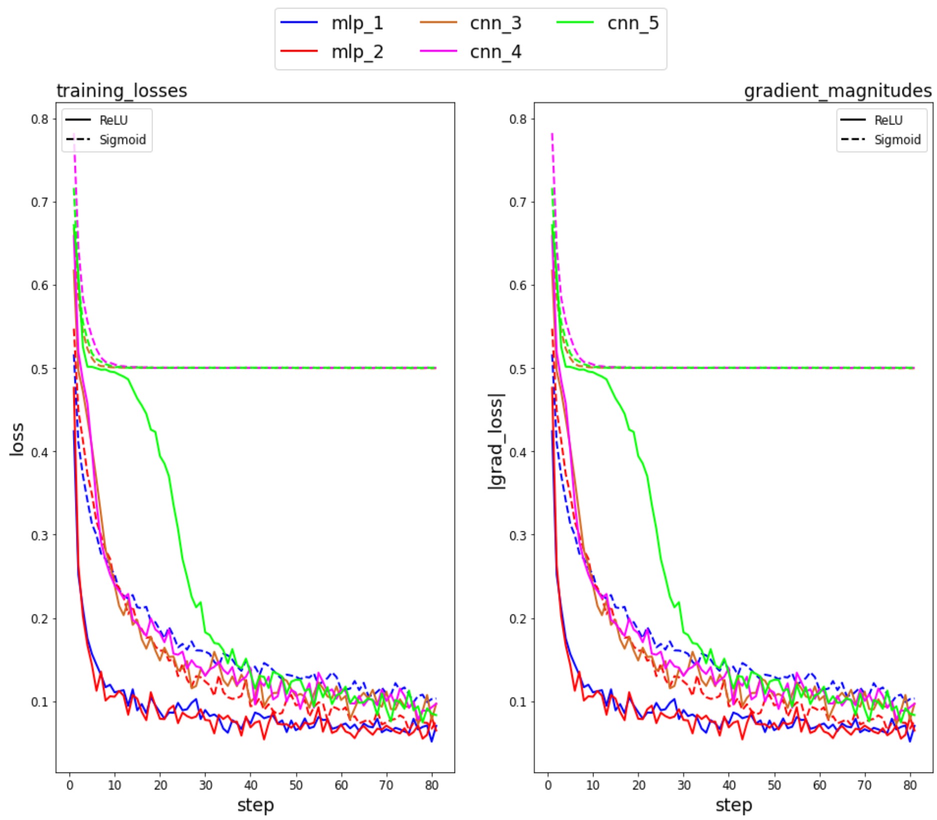

Figure 9: Shows the denoised version of the performance comparison graph in Figure 8.

3.1.2. Performance Comparison Plot for mlp_1 and mlp_2 Architectures

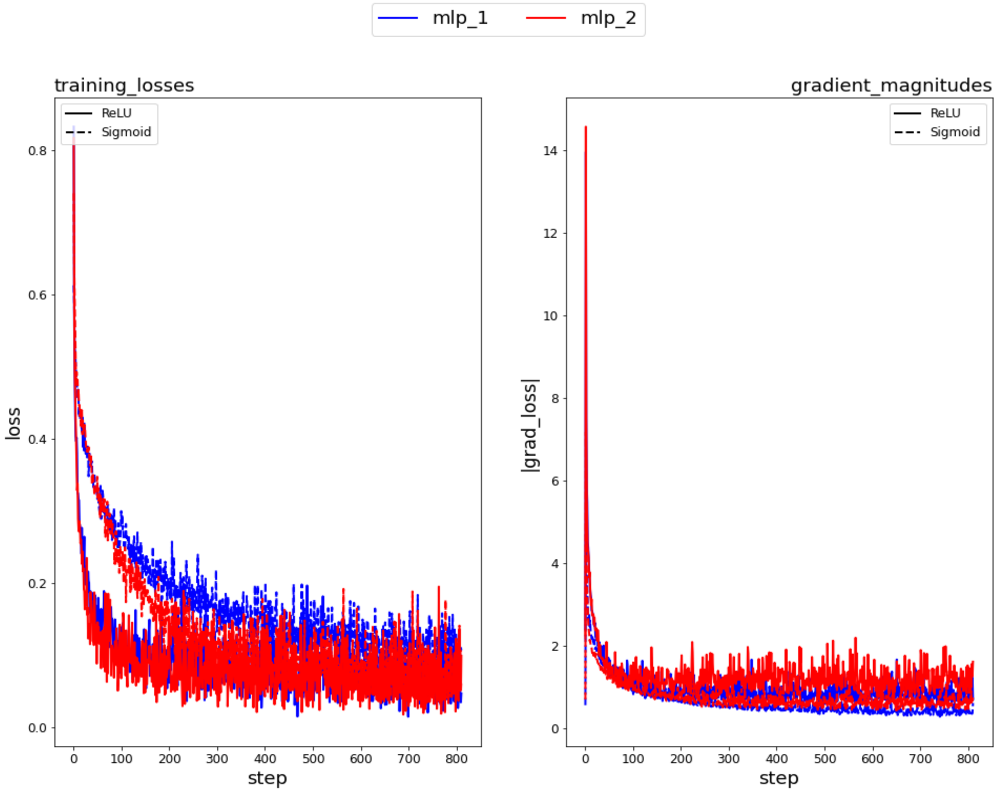

Figure 10: Shows mlp_1 and mlp_2 ANN architecture's training results on the same graph.

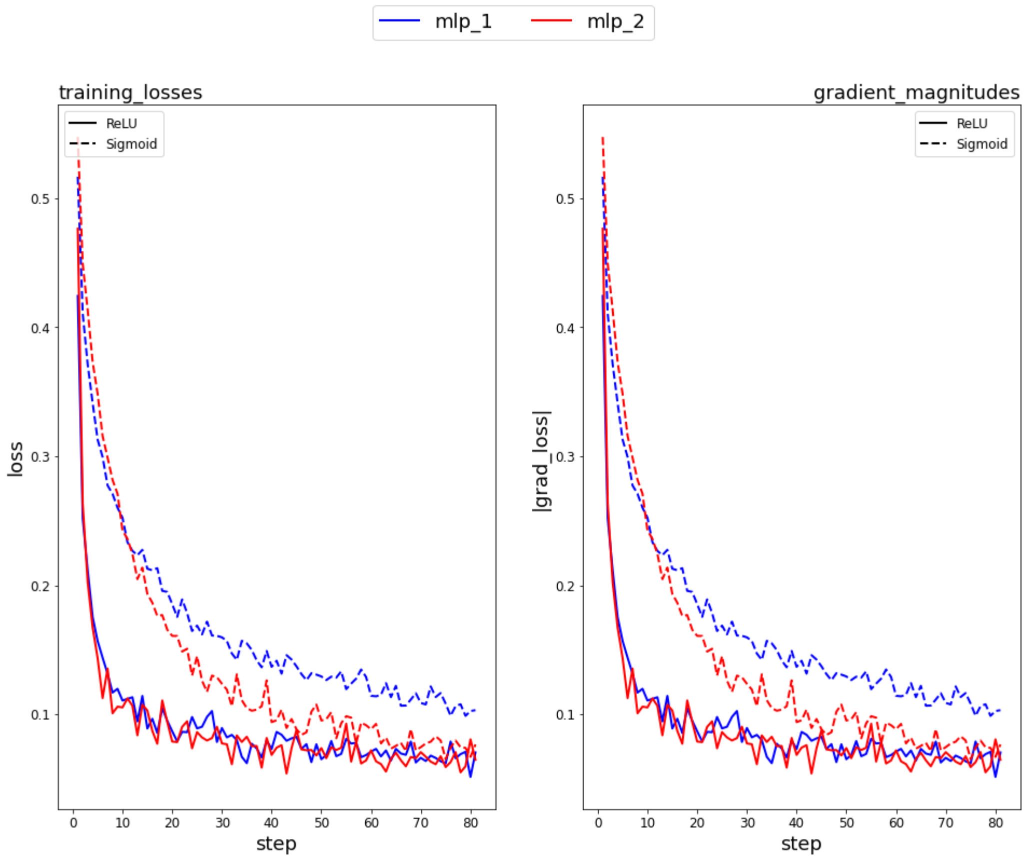

Figure 11: Shows the denoised version mlp_1 and mlp_2 training results in Figure 10.

3.1.3. Performance Comparison Plot for cnn_3, cnn_4 and cnn_5 Architectures

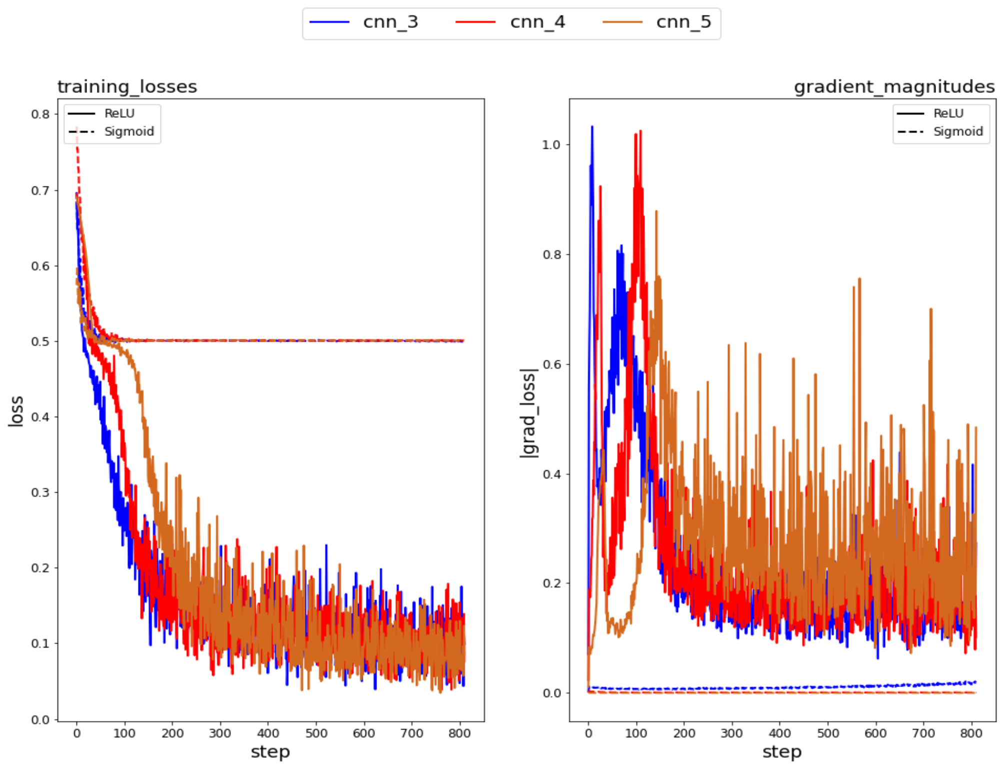

Figure 12: Shows CNN architectures' training results, on the same graph.

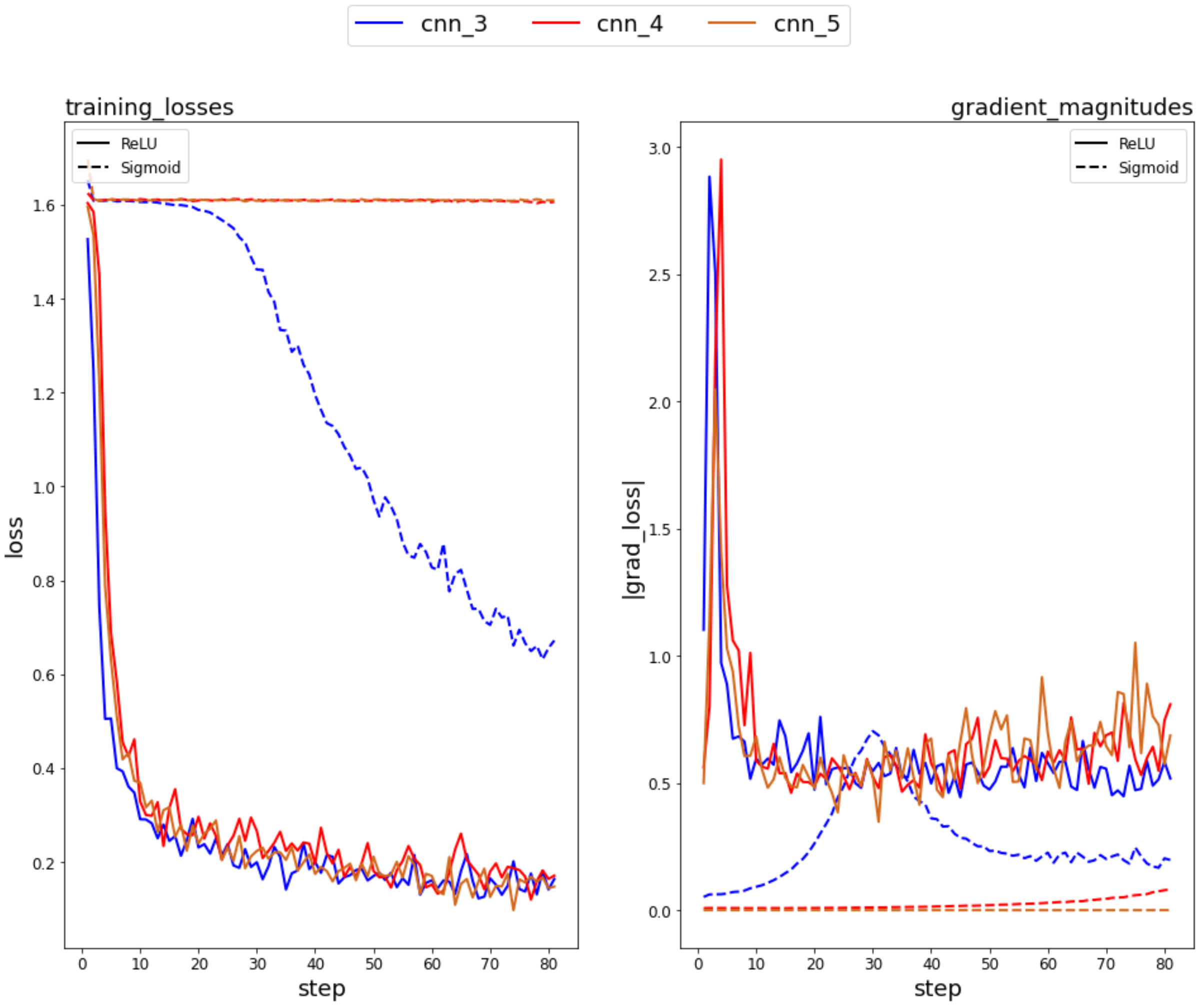

Figure 13: Shows the denoised version cnn_3, cnn_4 and cnn_5 training results in Figure 12.

3.2. Discussion

3.2.1. How is the gradient behavior in different architectures? What happens when depth increases? Why do you think that happens?

- If we consider Figure 10, as we add more dense layers to MLP architectures, gradient and loss converges faster to 0. The performance is also better as depth increases. Since the architecture has more parameters to train, it can arrange the weights such that the gradient magnitude is closer to 0.
- If we consider Figure 12, as we add more convolutional layers having the same activation function, we are facing more and more the vanishing gradient problem [2]. As we have more activation functions cascaded in the architecture, since the derivatives of the activation functions are usually low, the gradients also become low, and the training slows down. The problem is more visible for sigmoid activation function case, compared to ReLU, since derivative of the sigmoid is smaller than that of the ReLU.

3.2.2. What might happen if we do not scale the inputs to the range $[-1.0, 1.0]$?

- Since the input range will be higher, the gradient will have higher values for the same iteration number, oscillate more, converge slower.

4. Experimenting Learning Rate

4.1. Experimental Work

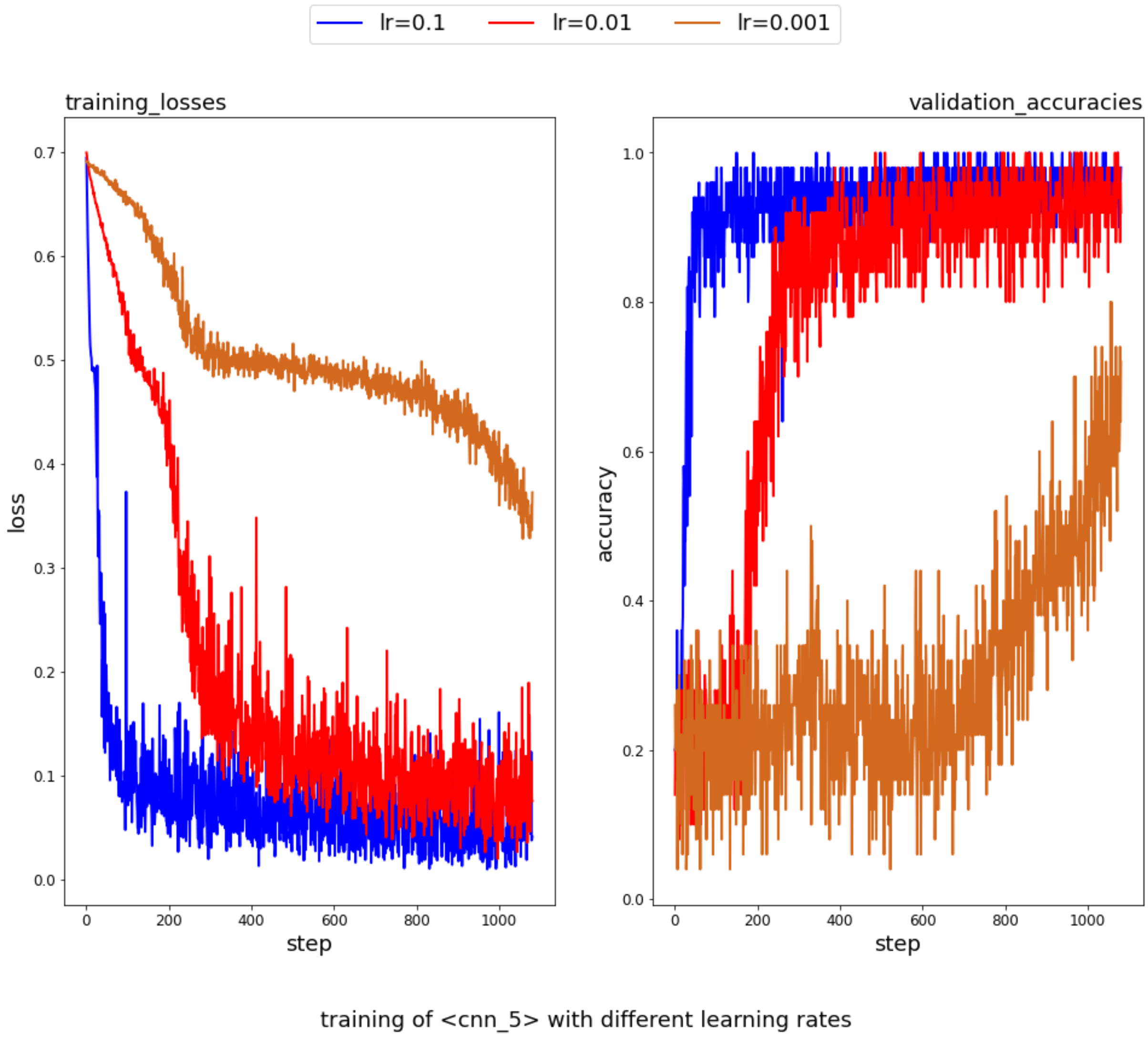

Figure 14: Shows the learning rate noisy experiment results.

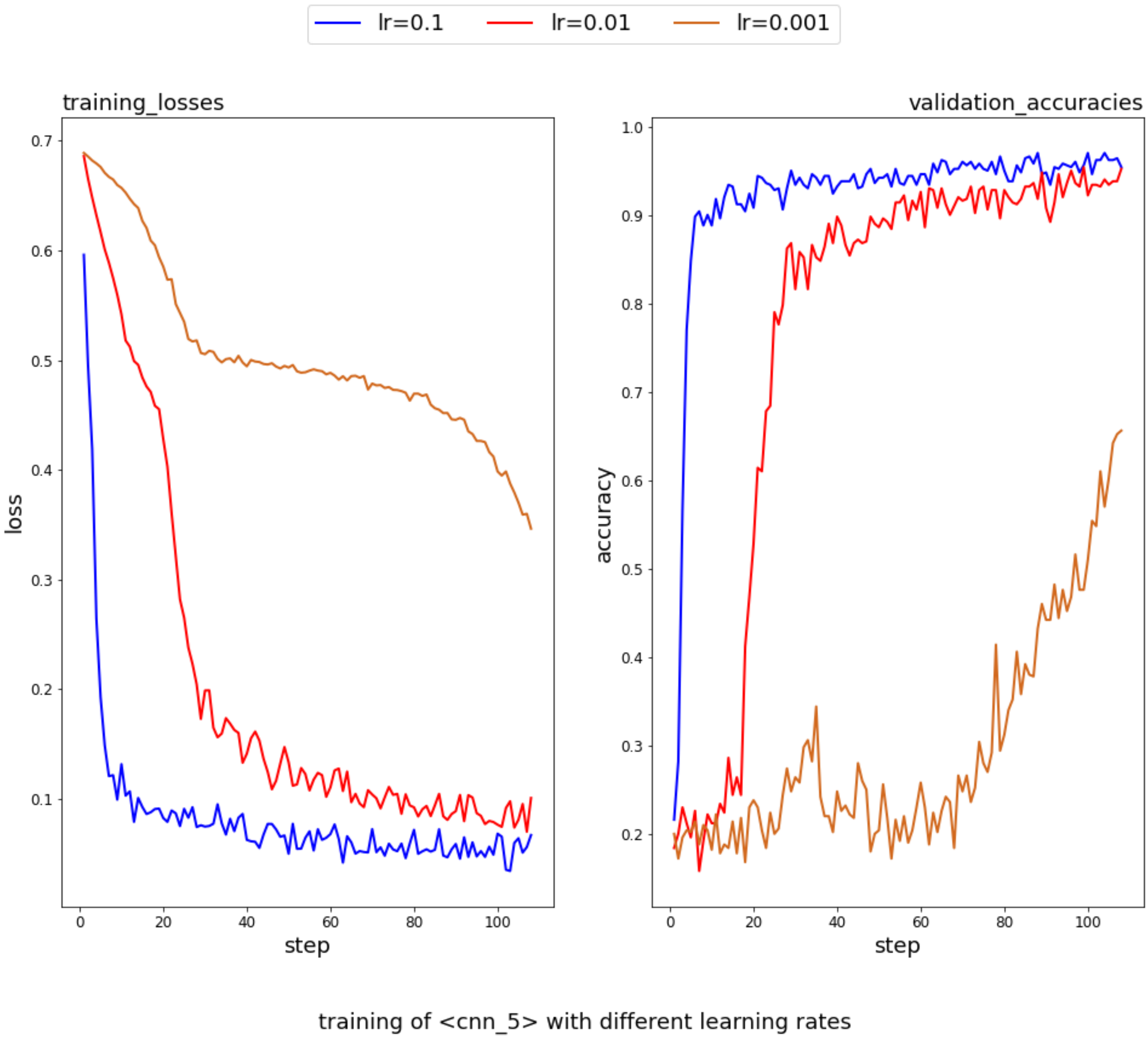

Figure 15: Shows the learning rate experiment results, after noise filtering.

It is easy to make observations on Figure 15. As can be seen on Figure 15, learning stops around step 25, which corresponds to 250 on Figure 14. Since we have batch size of 50 and 27000 training images, we have 540 steps in each epoch. Since we are recording once per 10 steps, one step in Figure 14 and Figure 15 corresponds to 10 steps of training.

Therefore, step 250 on Figure 14 means 2500 steps of training, corresponds to epoch: $floor\left(\frac{2500}{540}\right) = 5$

4.1.1. Scheculed Learning LR=0.01

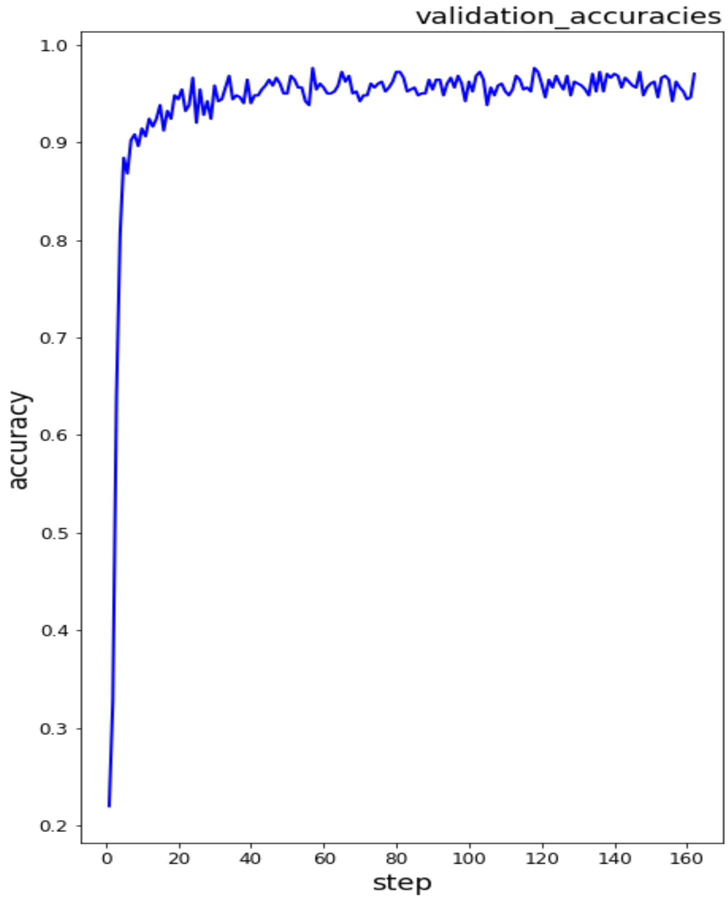

Figure 16: Shows the validation accuracy graph, obtained by training with LR=0.1 until epoch 6 and LR=0.01 after that.

If we look at closer and compare the accuracy curve of non-scheduled training and scheduled training, making small modifications on the resultant curves to compare better:

Training stops at step 40, which corresponds to epoch: $floor\left(\frac{4000}{540}\right) = 8$

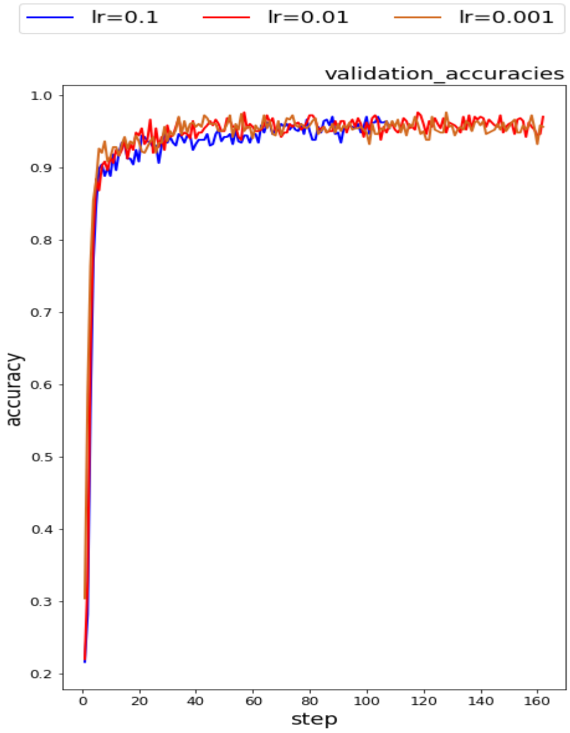

Figure 17: Shows the validation accuracy curves for constant learning rate (blue), two scheduled learning rates (red), and three scheduled learning rates (brown).

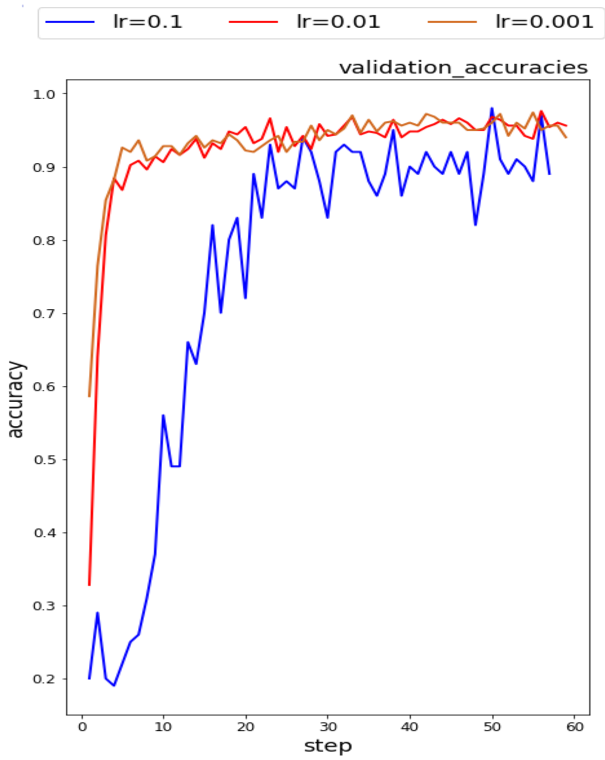

Figure 18: Shows the zoomed-in version of Figure 17.

4.2. Discussion

4.2.1. How Does the Learning Rate Affect the Convergence Speed?

Learning rate is a measure of how sensitive the weight updates will be to the loss gradient. If it is too high, we may miss the global minima point, which is set of weights that makes the gradient 0. If it is too small, then reaching global minima may take very long times.
For our experiment, considering Figure 15, increasing learning rate increase convergence speed.

4.2.2. How Does the Learning Rate Affect the Convergence to A Better Point?

If learning rate is too high, our model can oscillate around the global minima point, as can be seen Figure 18 for LR=0.1 case (blue curve). Small learning rate gives more stable accuracy curve.

4.2.3. Does Your Scheduled Learning Rate Method Work? In What Sense?

Considering Figure 18, when we apply scheduled learning, we get better accuracy and faster convergence. Also, although there is not so much difference, when we decreased the learning rate two times, we get faster convergence. Therefore, our scheduled learning rate model works.

4.2.4. Compare the Accuracy and Convergence Performance of Your Scheduled Learning Rate Method with Adam

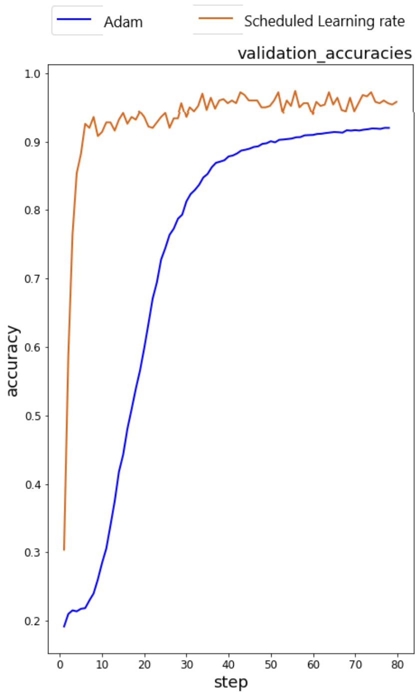

Figure 19: Shows the comparison of Adam (Figure 1, cnn_5, denoised) with scheduled learning rate (Figure 18).

As can clearly be seen in Figure 19, scheduled learning converges faster and gives better performance.

5. References

[1] https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-

fc88790d530d#:~:text=Number%20of%20parameters%20in%20a%20CONV%20layer%20would%20be%20%3A%20((,1)*number%20of%20filters).
[2] https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

6. Appendices

6.1. Appendix I: Code for Part-2

```
# import necessery packages
from sklearn.model_selection import train_test_split

import numpy as np
import tensorflow as tf
import os
import pickle
from random import randrange
from utils import part2Plots, visualizeWeights

# some parameters
INIT_LR = 1e-2  # initial learning rate
EPOCHS = 15  # epochs
BS = 50  # batch size
IMG_W = 28  # width of the images to be trained
IMG_H = 28  # height of the images to be trained
N_CLASSES = 5  # number of classes
DATA_PATH = 'dataset/'

# load the .npy formatted data
print("[INFO] loading data...")
train_img = np.load(DATA_PATH + 'train_images.npy')
train_lbl = np.load(DATA_PATH + 'train_labels.npy')
test_img = np.load(DATA_PATH + 'test_images.npy')
test_lbl = np.load(DATA_PATH + 'test_labels.npy')

# convert data to np.array float type
train_img = np.array(train_img, dtype="float32")
train_lbl = np.array(train_lbl, dtype="int")
test_img = np.array(test_img, dtype="float32")
test_lbl = np.array(test_lbl, dtype="int")

# preprocess images [0,255] -> [-1,1]
train_img = np.true_divide(train_img, 127.5) - 1
test_img = np.true_divide(test_img, 127.5) - 1

# partition the data into training and validation splits using 90% of
# the data for training and the remaining 10% for validation
(train_img, val_img, train_lbl, val_lbl) = train_test_split(train_img,
train_lbl,

test_size=0.10, stratify=train_lbl, random_state=42)
# create convolution formatted images
train_img_conv = train_img.reshape(-1, IMG_W, IMG_H, 1)
test_img_conv = test_img.reshape(-1, IMG_W, IMG_H, 1)
val_img_conv = val_img.reshape(-1, IMG_W, IMG_H, 1)

# perform one-hot encoding on the labels
train_lbl = tf.keras.utils.to_categorical(train_lbl, N_CLASSES)
test_lbl = tf.keras.utils.to_categorical(test_lbl, N_CLASSES)
```

```
val_lbl = tf.keras.utils.to_categorical(val_lbl, N_CLASSES)

# create the PredictionLayer
PredictionLayer = tf.keras.Sequential()
PredictionLayer.add(tf.keras.layers.Dense(units=5, activation='softmax'))


# ***************** MODEL CREATION FUNCTIONS ***************** #
def create_mlp_1():
    # construct mlp_1 model. [FC-64, ReLU] + PredictionLayer
    model_mlp_1 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
        tf.keras.layers.Dense(units=64, activation='relu'),  # FC-64
        PredictionLayer  # PredictionLayer
    ])

    # create binary cross entropy loss (one-hot encoding case)
    loss_mlp_1 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_mlp_1 = tf.keras.optimizers.SGD(learning_rate=INIT_LR)

    # compile model for training
    model_mlp_1.compile(optimizer=optimizer_mlp_1, loss=loss_mlp_1,
metrics=["accuracy"])

    print("--------------MLP_1 MODEL--------------")
    print(model_mlp_1.summary())

    return model_mlp_1


def create_mlp_2():
    # construct mlp_2 model. [FC-16, ReLU, FC-64(no bias)] +
PredictionLayer
    model_mlp_2 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
        tf.keras.layers.Dense(units=16, activation='relu'),  # FC-16,
ReLU,
        tf.keras.layers.Dense(units=64),  # FC-64
        PredictionLayer  # PredictionLayer
    ])
    print("--------------MLP_2 MODEL--------------")
    print(model_mlp_2.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_mlp_2 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_mlp_2 = tf.keras.optimizers.SGD(learning_rate=INIT_LR)

    # compile model for training
    model_mlp_2.compile(optimizer=optimizer_mlp_2, loss=loss_mlp_2,
metrics=["accuracy"])

    return model_mlp_2


def create_cnn_3():
    # construct cnn_3 model 'cnn 3' : [Conv-3×3×16, ReLU, Conv-7×7×8,
ReLU,
```

```
    # MaxPool-2×2, Conv-5×5×16, MaxPool-2×2,
    model_cnn_3 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=8, kernel_size=7,
activation='relu'),  # Conv-7×7×8, ReLU,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.Conv2D(filters=16, kernel_size=5,
activation='relu'),  # Conv-7×7×8, ReLU,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
    ])
    print("--------------CNN_3 MODEL--------------")
    print(model_cnn_3.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_cnn_3 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_cnn_3 = tf.keras.optimizers.SGD(learning_rate=INIT_LR)

    # compile model for training
    model_cnn_3.compile(optimizer=optimizer_cnn_3, loss=loss_cnn_3,
metrics=["accuracy"])

    return model_cnn_3


def create_cnn_4():
    # construct cnn_4 model 'cnn 4' : ['cnn 3' : [Conv-3×3×16, ReLU,
    # Conv-5×5×8, ReLU, Conv-3×3×8, ReLU, MaxPool-2×2, Conv-5×5×16, ReLU,
MaxPool-2×2,
    # GlobalAvgPool] + PredictionLayer
    model_cnn_4 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=8, kernel_size=5,
activation='relu'),  # Conv-5×5×8, ReLU,
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.Conv2D(filters=16, kernel_size=5,
activation='relu'),  # Conv-7×7×8, ReLU,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
    ])
    print("--------------CNN_4 MODEL--------------")
    print(model_cnn_4.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_cnn_4 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_cnn_4 = tf.keras.optimizers.SGD(learning_rate=INIT_LR)
```

```
    # compile model for training
    model_cnn_4.compile(optimizer=optimizer_cnn_4, loss=loss_cnn_4,
metrics=["accuracy"])

    return model_cnn_4


def create_cnn_5():
    # construct cnn_5 model. 'cnn 5' : [Conv-3×3×16, ReLU, Conv-3×3×8,
ReLU, Conv-3×3×8, ReLU,
    # Conv-3×3×8, ReLU, MaxPool-2×2, Conv-3×3×16, ReLU, Conv-3×3×16,
ReLU, MaxPool-2×2,
    # GlobalAvgPool] + PredictionLayer
    model_cnn_5 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU,
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
    ])
    print("--------------CNN_5 MODEL--------------")
    print(model_cnn_5.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_cnn_5 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_cnn_5 = tf.keras.optimizers.SGD(learning_rate=INIT_LR)

    # compile model for training
    model_cnn_5.compile(optimizer=optimizer_cnn_5, loss=loss_cnn_5,
metrics=["accuracy"])

    return model_cnn_5

    # create model by name


def create_model(model_name):
    if model_name == "mlp_1":
        return create_mlp_1()
    elif model_name == "mlp_2":
        return create_mlp_2()
    elif model_name == "cnn_3":
        return create_cnn_3()
    elif model_name == "cnn_4":
        return create_cnn_4()
    elif model_name == "cnn_5":
```

```
        return create_cnn_5()
    else:
        return None


# load and save functions
def save_obj(obj, name):
    with open('part2/results/part2_' + name + '.pkl', 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def load_obj(name):
    with open('part2/results/part2_' + name + '.pkl', 'rb') as f:
        return pickle.load(f)


# my fit function that uses train_on_batch()
def my_fit(model_name, x_train, y_train, x_test, y_test,
           x_val, y_val, iteration=1, epoches=15):
    # declare and initialize some parameters
    split_size = len(y_train) // BS
    split_size_val = len(y_val) // BS
    length = x_train.shape[0]
    losses = []
    train_accs = []
    test_accs = []
    val_accs = []
    weights = []
    # split validation dataset into bacthes
    val_xb = np.array_split(x_val, split_size_val)
    val_yb = np.array_split(y_val, split_size_val)
    for i in range(iteration):
        # in each iteration, create a new model
        model = create_model(model_name)
        # at each iteration, we have seperate loss and accuracy curves
        loss = []
        train_acc = []
        val_acc = []
        print("------------", model_name, " iteration: ", (i + 1), "-----
-------")
        for j in range(epoches):
            print("epoch: " + str(j) + " ----------------------------> ")

            # suffle the training set
            idxs = np.arange(0, length)
            np.random.shuffle(idxs)

            x_train = x_train[idxs]
            y_train = y_train[idxs]

            # extract batches
            train_xb = np.array_split(x_train, split_size)
            train_yb = np.array_split(y_train, split_size)

            # train the batches
            index = list(range(split_size))
            for i in index:
                results = model.train_on_batch(train_xb[i], train_yb[i])
                if i % 10 == 0:  # every 10 steps
                    # record the loss
                    loss.append(results[0])
```

```
                    # record the accuracy
                    train_acc.append(results[1])
                    # record the validaditon accuracy
                    rnd_idx = randrange(split_size_val)
                    acc_val = model.evaluate(val_xb[rnd_idx],
val_yb[rnd_idx])[1]
                    val_acc.append(acc_val)

        # record the loss and accuracy curves of each trial
        losses.append(np.array(loss))
        train_accs.append(np.array(train_acc))
        val_accs.append(np.array(val_acc))
        weights.append(model.trainable_weights[0].numpy())
        # Compute test accuracy
        test_accs.append(model.evaluate(x_test, y_test)[1])

    # convert list of np arrays to np arrays
    losses = np.array(losses)
    train_accs = np.array(train_accs)
    test_accs = np.array(test_accs)
    val_accs = np.array(val_accs)
    weights = np.array(weights)

    best_idx = np.argmax(test_accs)
    dic = {
        "name": model_name,
        "loss_curve": np.mean(losses, axis=0),
        "train_acc_curve": np.mean(train_accs, axis=0),
        "val_acc_curve": np.mean(val_accs, axis=0),
        "test_acc": test_accs[best_idx],
        "weights": weights[best_idx]
    }

    return dic


# train mlp_1 model and save the results
dic1 = my_fit("mlp_1", train_img, train_lbl, test_img, test_lbl, val_img,
val_lbl)
save_obj(dic1, "mlp_1")

# train mlp_2 model and save the results
dic2 = my_fit("mlp_2", train_img, train_lbl, test_img, test_lbl, val_img,
val_lbl)
save_obj(dic2, "mlp_2")

# train cnn_3 model and save the results
dic3 = my_fit("cnn_3", train_img_conv, train_lbl, test_img_conv,
test_lbl,
              val_img_conv, val_lbl)
save_obj(dic3, "cnn_3")

# train cnn_4 model and save the results
dic4 = my_fit("cnn_4", train_img_conv, train_lbl, test_img_conv,
test_lbl,
              val_img_conv, val_lbl)
save_obj(dic4, "cnn_4")

# train cnn_5 model and save the results
dic5 = my_fit("cnn_5", train_img_conv, train_lbl, test_img_conv,
test_lbl,
```

```
                val_img_conv, val_lbl)
save_obj(dic5, "cnn_5")

# draw the curves
results = [dic1, dic2, dic3, dic4, dic5]
part2Plots(results, save_dir='part2/plots/', filename='part2_plot',
show_plot=True)

# visualize the weights
visualizeWeights(dic1['weights'], save_dir='part2/plots',
filename="mlp_1_weights")
visualizeWeights(dic2['weights'], save_dir='part2/plots',
filename="mlp_2_weights")
visualizeWeights(dic3['weights'], save_dir='part2/plots',
filename="cnn_3_weights")
visualizeWeights(dic4['weights'], save_dir='part2/plots',
filename="cnn_4_weights")
visualizeWeights(dic5['weights'], save_dir='part2/plots',
filename="cnn_5_weights")
```

6.2. Appendix II: Code for Part-3

```
# import necessery packages
from sklearn.model_selection import train_test_split

import numpy as np
import tensorflow as tf
import pickle

from utils import part3Plots

# INIT_LR = 1e-4 # initial learning rate
EPOCHS = 15  # epochs
BS = 50  # batch size
IMG_W = 28  # width of the images to be trained
IMG_H = 28  # height of the images to be trained
N_CLASSES = 5  # number of classes
IMG_W = 28
IMG_H = 28

DATA_PATH = 'dataset/'

# load the .npy formatted data
print("[INFO] loading data...")
train_img = np.load(DATA_PATH + 'train_images.npy')
train_lbl = np.load(DATA_PATH + 'train_labels.npy')
test_img = np.load(DATA_PATH + 'test_images.npy')
test_lbl = np.load(DATA_PATH + 'test_labels.npy')

# convert data to np.array float type
train_img = np.array(train_img, dtype="float32")
train_lbl = np.array(train_lbl, dtype="int")
test_img = np.array(test_img, dtype="float32")
test_lbl = np.array(test_lbl, dtype="int")

# preprocess images [0,255] -> [-1,1]
train_img = (train_img / 127.5) - 1
```

```
test_img = (test_img / 127.5) - 1

# partition the data into training and validation splits using 90% of
# the data for training and the remaining 10% for validation
(train_img, val_img, train_lbl, val_lbl) = train_test_split(train_img,
train_lbl,

test_size=0.10, stratify=train_lbl, random_state=42)

# reformat images for cnn
cnn_train_img = train_img.reshape(-1, IMG_W, IMG_H, 1)

# perform one-hot encoding on the labels
train_lbl = tf.keras.utils.to_categorical(train_lbl, N_CLASSES)
test_lbl = tf.keras.utils.to_categorical(test_lbl, N_CLASSES)
val_lbl = tf.keras.utils.to_categorical(val_lbl, N_CLASSES)

# create the PredictionLayer
PredictionLayer = tf.keras.Sequential()
PredictionLayer.add(tf.keras.layers.Dense(units=5, activation='softmax'))

# construct mlp_1 model. [FC-64, ReLU] + PredictionLayer
model_mlp_1_relu = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
    tf.keras.layers.Dense(units=64, activation='relu'),  # FC-64
    PredictionLayer  # PredictionLayer
])
print("--------------MLP_1 MODEL(RELU)--------------")
print(model_mlp_1_relu.summary())

model_mlp_1_sigm = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
    tf.keras.layers.Dense(units=64, activation='sigmoid'),  # FC-64
    PredictionLayer  # PredictionLayer
])
print("--------------MLP_1 MODEL(SIGMOID)--------------")
print(model_mlp_1_sigm.summary())

# construct mlp_2 model. [FC-16, ReLU, FC-64(no bias)] + PredictionLayer
model_mlp_2_relu = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
    tf.keras.layers.Dense(units=16, activation='relu'),  # FC-16, ReLU,
    tf.keras.layers.Dense(units=64),  # FC-64
    PredictionLayer  # PredictionLayer
])
print("--------------MLP_2 MODEL(RELU)--------------")
print(model_mlp_2_relu.summary())

model_mlp_2_sigm = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
    tf.keras.layers.Dense(units=16, activation='sigmoid'),  # FC-16,
ReLU,
    tf.keras.layers.Dense(units=64),  # FC-64
    PredictionLayer  # PredictionLayer
])
print("--------------MLP_2 MODEL(SIGMOID)--------------")
print(model_mlp_2_sigm.summary())

# construct cnn_3 model 'cnn 3' : [Conv-3×3×16, ReLU, Conv-7×7×8, ReLU,
# MaxPool-2×2, Conv-5×5×16, MaxPool-2×2,
model_cnn_3_relu = tf.keras.Sequential([
```

```
    tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
    tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation='relu'),
# Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=8, kernel_size=7, activation='relu'),
# Conv-7×7×8, ReLU,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.Conv2D(filters=16, kernel_size=5, activation='relu'),
# Conv-5×5×16, ReLU,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
])
print("--------------CNN_3 MODEL(RELU)--------------")
print(model_cnn_3_relu.summary())

model_cnn_3_sigm = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
    tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=8, kernel_size=7,
activation='sigmoid'),  # Conv-7×7×8, ReLU,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.Conv2D(filters=16, kernel_size=5,
activation='sigmoid'),  # Conv-7×7×8, ReLU,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
])
print("--------------CNN_3 MODEL(SIGM)--------------")
print(model_cnn_3_sigm.summary())

# construct cnn_4 model 'cnn 4' : ['cnn 3' : [Conv-3×3×16, ReLU,
# Conv-5×5×8, ReLU, Conv-3×3×8, ReLU, MaxPool-2×2, Conv-5×5×16, ReLU,
MaxPool-2×2,
# GlobalAvgPool] + PredictionLayer
model_cnn_4_relu = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
    tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation='relu'),
# Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=8, kernel_size=5, activation='relu'),
# Conv-5×5×8, ReLU,
    tf.keras.layers.Conv2D(filters=8, kernel_size=3, activation='relu'),
# Conv-3×3×8, ReLU
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.Conv2D(filters=16, kernel_size=5, activation='relu'),
# Conv-5×5×16, ReLU,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
])
print("--------------CNN_4 MODEL(RELU)--------------")
print(model_cnn_4_relu.summary())

model_cnn_4_sigm = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
    tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×16, Relu,
```

```
    tf.keras.layers.Conv2D(filters=8, kernel_size=5,
activation='sigmoid'),  # Conv-5×5×8, ReLU,
    tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×8, ReLU
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.Conv2D(filters=16, kernel_size=5,
activation='sigmoid'),  # Conv-7×7×8, ReLU,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
])
print("--------------CNN_4 MODEL(SIGMOID)--------------")
print(model_cnn_4_sigm.summary())

# construct cnn_5 model. 'cnn 5' : [Conv-3×3×16, ReLU, Conv-3×3×8, ReLU,
Conv-3×3×8, ReLU,
# Conv-3×3×8, ReLU, MaxPool-2×2, Conv-3×3×16, ReLU, Conv-3×3×16, ReLU,
MaxPool-2×2,
# GlobalAvgPool] + PredictionLayer
model_cnn_5_relu = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
    tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation='relu'),
# Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=8, kernel_size=3, activation='relu'),
# Conv-5×5×8, ReLU,
    tf.keras.layers.Conv2D(filters=8, kernel_size=3, activation='relu'),
# Conv-3×3×8, ReLU
    tf.keras.layers.Conv2D(filters=8, kernel_size=3, activation='relu'),
# Conv-3×3×8, ReLU
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation='relu'),
# Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation='relu'),
# Conv-3×3×16, Relu,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
])
print("--------------CNN_5 MODEL (RELU)--------------")
print(model_cnn_5_relu.summary())

model_cnn_5_sigm = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
    tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='sigmoid'),  # Conv-5×5×8, ReLU,
    tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×8, ReLU
    tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×8, ReLU
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×16, Relu,
    tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='sigmoid'),  # Conv-3×3×16, Relu,
    tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
    tf.keras.layers.GlobalAveragePooling2D(),
```

```
    tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
])
print("--------------CNN_5 MODEL(SIGM)--------------")
print(model_cnn_5_sigm.summary())

# create binary cross entropy loss (one-hot encoding case)
loss_mlp_1_relu = tf.keras.losses.CategoricalCrossentropy()
loss_mlp_2_relu = tf.keras.losses.CategoricalCrossentropy()
loss_cnn_3_relu = tf.keras.losses.CategoricalCrossentropy()
loss_cnn_4_relu = tf.keras.losses.CategoricalCrossentropy()
loss_cnn_5_relu = tf.keras.losses.CategoricalCrossentropy()

# create optimizer
optimizer_mlp_1_relu = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_mlp_2_relu = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_cnn_3_relu = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_cnn_4_relu = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_cnn_5_relu = tf.keras.optimizers.SGD(learning_rate=0.01)

# compile model for training
model_mlp_1_relu.compile(optimizer=optimizer_mlp_1_relu,
loss=loss_mlp_1_relu, metrics=["accuracy"])
model_mlp_2_relu.compile(optimizer=optimizer_mlp_2_relu,
loss=loss_mlp_2_relu, metrics=["accuracy"])
model_cnn_3_relu.compile(optimizer=optimizer_cnn_3_relu,
loss=loss_cnn_3_relu, metrics=["accuracy"])
model_cnn_4_relu.compile(optimizer=optimizer_cnn_4_relu,
loss=loss_cnn_4_relu, metrics=["accuracy"])
model_cnn_5_relu.compile(optimizer=optimizer_cnn_5_relu,
loss=loss_cnn_5_relu, metrics=["accuracy"])

# create binary cross entropy loss (one-hot encoding case)
loss_mlp_1_sigm = tf.keras.losses.CategoricalCrossentropy()
loss_mlp_2_sigm = tf.keras.losses.CategoricalCrossentropy()
loss_cnn_3_sigm = tf.keras.losses.CategoricalCrossentropy()
loss_cnn_4_sigm = tf.keras.losses.CategoricalCrossentropy()
loss_cnn_5_sigm = tf.keras.losses.CategoricalCrossentropy()

# create optimizer
optimizer_mlp_1_sigm = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_mlp_2_sigm = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_cnn_3_sigm = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_cnn_4_sigm = tf.keras.optimizers.SGD(learning_rate=0.01)
optimizer_cnn_5_sigm = tf.keras.optimizers.SGD(learning_rate=0.01)

# compile model for training
model_mlp_1_sigm.compile(optimizer=optimizer_mlp_1_sigm,
loss=loss_mlp_1_sigm, metrics=["accuracy"])
model_mlp_2_sigm.compile(optimizer=optimizer_mlp_2_sigm,
loss=loss_mlp_2_sigm, metrics=["accuracy"])
model_cnn_3_sigm.compile(optimizer=optimizer_cnn_3_sigm,
loss=loss_cnn_3_sigm, metrics=["accuracy"])
model_cnn_4_sigm.compile(optimizer=optimizer_cnn_4_sigm,
loss=loss_cnn_4_sigm, metrics=["accuracy"])
model_cnn_5_sigm.compile(optimizer=optimizer_cnn_5_sigm,
loss=loss_cnn_5_sigm, metrics=["accuracy"])


# training function
def my_fit(model, x_train, y_train, iteration=1, epoches=15):
```

```
    split_size = len(y_train) // BS
    length = x_train.shape[0]
    weights = model.trainable_weights[0].numpy()
    losses = []
    grads = []
    for i in range(iteration):
        print('iteration: ', (i + 1))
        for j in range(epoches):
            print("epoch: " + str(j) + " ----------------------------> ")

            idxs = np.arange(0, length)
            np.random.shuffle(idxs)

            x_train = x_train[idxs]
            y_train = y_train[idxs]
            xb = np.array_split(x_train, split_size)
            yb = np.array_split(y_train, split_size)

            acc = 0
            index = list(range(split_size))
            for i in index:
                results = model.train_on_batch(xb[i], yb[i])
                acc += results[1]
                if i % 10 == 0:
                    losses.append(results[0])
                    weights_new = model.trainable_weights[0].numpy()
                    grad = (weights_new - weights) / 0.01
                    grads.append(np.linalg.norm(grad))
                    weights = weights_new

            acc = acc / split_size
            print("accuracy: ", acc)

    dictionary = {
        "losses": losses,
        "grads": grads,
    }

    return dictionary


def save_obj(obj, name):
    with open('part3/results/part3_' + name + '.pkl', 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def load_obj(name):
    with open('part3/results/part3_' + name + '.pkl', 'rb') as f:
        return pickle.load(f)


dic_mlp_1_relu = my_fit(model_mlp_1_relu, train_img, train_lbl)
dic_mlp_1_sigm = my_fit(model_mlp_1_sigm, train_img, train_lbl)
dic_mlp_2_relu = my_fit(model_mlp_2_relu, train_img, train_lbl)
dic_mlp_2_sigm = my_fit(model_mlp_2_sigm, train_img, train_lbl)
dic_cnn_3_relu = my_fit(model_cnn_3_relu, cnn_train_img, train_lbl)
dic_cnn_3_sigm = my_fit(model_cnn_3_sigm, cnn_train_img, train_lbl)
dic_cnn_4_relu = my_fit(model_cnn_4_relu, cnn_train_img, train_lbl)
dic_cnn_4_sigm = my_fit(model_cnn_4_sigm, cnn_train_img, train_lbl)
dic_cnn_5_relu = my_fit(model_cnn_5_relu, cnn_train_img, train_lbl)
dic_cnn_5_sigm = my_fit(model_cnn_5_sigm, cnn_train_img, train_lbl)
```

```python
def to_result_dic(name, dic_relu, dic_sigm):
    result = {
        "name": name,
        "relu_loss_curve": dic_relu["losses"],
        "sigmoid_loss_curve": dic_sigm["losses"],
        "relu_grad_curve": dic_relu["grads"],
        "sigmoid_grad_curve": dic_sigm["grads"]
    }
    return result


# convert the results to a suitable format to print
dic_mlp_1 = to_result_dic("mlp_1", dic_mlp_1_relu, dic_mlp_1_sigm)
dic_mlp_2 = to_result_dic("mlp_2", dic_mlp_2_relu, dic_mlp_2_sigm)
dic_cnn_3 = to_result_dic("cnn_3", dic_cnn_3_relu, dic_cnn_3_sigm)
dic_cnn_4 = to_result_dic("cnn_4", dic_cnn_4_relu, dic_cnn_4_sigm)
dic_cnn_5 = to_result_dic("cnn_5", dic_cnn_5_relu, dic_cnn_5_sigm)
save_obj(dic_mlp_1, "mlp_1")
save_obj(dic_mlp_2, "mlp_2")
save_obj(dic_cnn_3, "cnn_3")
save_obj(dic_cnn_4, "cnn_4")
save_obj(dic_cnn_5, "cnn_5")


# Reduces the shape (x,) to (x/10,) by taking average of each 10 sample.
def reduce_graph_noise(dictionary, avr=10):
    result = {
        "name": dictionary["name"],
        "relu_loss_curve":
np.mean(np.array(dictionary["relu_loss_curve"]).reshape(-1, avr),
axis=1),
        "sigmoid_loss_curve":
np.mean(np.array(dictionary["sigmoid_loss_curve"]).reshape(-1, avr),
axis=1),
        "relu_grad_curve":
np.mean(np.array(dictionary["relu_grad_curve"]).reshape(-1, avr),
axis=1),
        "sigmoid_grad_curve":
np.mean(np.array(dictionary["sigmoid_grad_curve"]).reshape(-1, avr),
axis=1)
    }
    return result


# reduce the noise of curves
dic_mlp_1 = reduce_graph_noise(dic_mlp_1)
dic_mlp_2 = reduce_graph_noise(dic_mlp_2)
dic_cnn_3 = reduce_graph_noise(dic_cnn_3)
dic_cnn_4 = reduce_graph_noise(dic_cnn_4)
dic_cnn_5 = reduce_graph_noise(dic_cnn_5)

results = [ dic_mlp_1, dic_mlp_2, dic_cnn_3, dic_cnn_4, dic_cnn_5]
part3Plots(results, save_dir='part3/plots/', filename='part3_graph',
show_plot=True)
```

6.3. Appendix III: Code for Part-4

```
# import necessary packages
from sklearn.model_selection import train_test_split

import numpy as np
import pickle
import tensorflow as tf
import os
from random import randrange
from utils import part4Plots

# some parameters
LR_1 = 1e-1  # 0.1
LR_2 = 1e-2  # 0.01
LR_3 = 1e-3  # 0.001
LR = [LR_1, LR_2, LR_3]
EPOCHS = 20  # epochs
BS = 50  # batch size
IMG_W = 28  # width of the images to be trained
IMG_H = 28  # height of the images to be trained
N_CLASSES = 5  # number of classes

DATA_PATH = 'dataset/'

# load the .npy formatted data
print("[INFO] loading data...")
train_img = np.load(DATA_PATH + 'train_images.npy')
train_lbl = np.load(DATA_PATH + 'train_labels.npy')
test_img = np.load(DATA_PATH + 'test_images.npy')
test_lbl = np.load(DATA_PATH + 'test_labels.npy')

# convert data to np.array float type
train_img = np.array(train_img, dtype="float32")
train_lbl = np.array(train_lbl, dtype="int")
test_img = np.array(test_img, dtype="float32")
test_lbl = np.array(test_lbl, dtype="int")

# preprocess images [0,255] -> [-1,1]
train_img = np.true_divide(train_img, 127.5) - 1
test_img = np.true_divide(test_img, 127.5) - 1

# partition the data into training and validation splits using 90% of
# the data for training and the remaining 10% for validation
(train_img, val_img, train_lbl, val_lbl) = train_test_split(train_img,
train_lbl,

test_size=0.10, stratify=train_lbl, random_state=42)
# create convolution format
train_img_conv = train_img.reshape(-1, IMG_W, IMG_H, 1)
test_img_conv = test_img.reshape(-1, IMG_W, IMG_H, 1)
val_img_conv = val_img.reshape(-1, IMG_W, IMG_H, 1)

# perform one-hot encoding on the labels
train_lbl = tf.keras.utils.to_categorical(train_lbl, N_CLASSES)
test_lbl = tf.keras.utils.to_categorical(test_lbl, N_CLASSES)
val_lbl = tf.keras.utils.to_categorical(val_lbl, N_CLASSES)

# create the PredictionLayer
PredictionLayer = tf.keras.Sequential()
PredictionLayer.add(tf.keras.layers.Dense(units=5, activation='softmax'))
```

```
def create_mlp_1(learning_rate=LR[0]):
    # construct mlp_1 model. [FC-64, ReLU] + PredictionLayer
    model_mlp_1 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
        tf.keras.layers.Dense(units=64, activation='relu'),  # FC-64
        PredictionLayer  # PredictionLayer
    ])

    # create binary cross entropy loss (one-hot encoding case)
    loss_mlp_1 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_mlp_1 =
tf.keras.optimizers.SGD(learning_rate=learning_rate)

    # compile model for training
    model_mlp_1.compile(optimizer=optimizer_mlp_1, loss=loss_mlp_1,
metrics=["accuracy"])

    print("--------------MLP_1 MODEL--------------")
    print(model_mlp_1.summary())

    return model_mlp_1


def create_mlp_2(learning_rate=LR[0]):
    # construct mlp_2 model. [FC-16, ReLU, FC-64(no bias)] +
PredictionLayer
    model_mlp_2 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W * IMG_H,)),  # input layer
        tf.keras.layers.Dense(units=16, activation='relu'),  # FC-16,
ReLU,
        tf.keras.layers.Dense(units=64),  # FC-64
        PredictionLayer  # PredictionLayer
    ])
    print("--------------MLP_2 MODEL--------------")
    print(model_mlp_2.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_mlp_2 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_mlp_2 =
tf.keras.optimizers.SGD(learning_rate=learning_rate)

    # compile model for training
    model_mlp_2.compile(optimizer=optimizer_mlp_2, loss=loss_mlp_2,
metrics=["accuracy"])

    return model_mlp_2


def create_cnn_3(learning_rate=LR[0]):
    # construct cnn_3 model 'cnn 3' : [Conv-3×3×16, ReLU, Conv-7×7×8,
ReLU,
    # MaxPool-2×2, Conv-5×5×16, MaxPool-2×2,
    model_cnn_3 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
```

```
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=8, kernel_size=7,
activation='relu'),  # Conv-7×7×8, ReLU,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.Conv2D(filters=16, kernel_size=5,
activation='relu'),  # Conv-7×7×8, ReLU,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
    ])
    print("--------------CNN_3 MODEL--------------")
    print(model_cnn_3.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_cnn_3 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_cnn_3 =
tf.keras.optimizers.SGD(learning_rate=learning_rate)

    # compile model for training
    model_cnn_3.compile(optimizer=optimizer_cnn_3, loss=loss_cnn_3,
metrics=["accuracy"])

    return model_cnn_3


def create_cnn_4(learning_rate=LR[0]):
    # construct cnn_4 model 'cnn 4' : ['cnn 3' : [Conv-3×3×16, ReLU,
    # Conv-5×5×8, ReLU, Conv-3×3×8, ReLU, MaxPool-2×2, Conv-5×5×16, ReLU,
MaxPool-2×2,
    # GlobalAvgPool] + PredictionLayer
    model_cnn_4 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=8, kernel_size=5,
activation='relu'),  # Conv-5×5×8, ReLU,
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.Conv2D(filters=16, kernel_size=5,
activation='relu'),  # Conv-7×7×8, ReLU,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
    ])
    print("--------------CNN_4 MODEL--------------")
    print(model_cnn_4.summary())

    # create binary cross entropy loss (one-hot encoding case)
    loss_cnn_4 = tf.keras.losses.CategoricalCrossentropy()

    # create optimizer
    optimizer_cnn_4 =
tf.keras.optimizers.SGD(learning_rate=learning_rate)

    # compile model for training
```

```
    model_cnn_4.compile(optimizer=optimizer_cnn_4, loss=loss_cnn_4,
metrics=["accuracy"])

    return model_cnn_4


def create_cnn_5(learning_rate=LR[0], loss=None, optimizer=None):
    # construct cnn_5 model. 'cnn 5' : [Conv-3×3×16, ReLU, Conv-3×3×8,
ReLU, Conv-3×3×8, ReLU,
    # Conv-3×3×8, ReLU, MaxPool-2×2, Conv-3×3×16, ReLU, Conv-3×3×16,
ReLU, MaxPool-2×2,
    # GlobalAvgPool] + PredictionLayer
    model_cnn_5 = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(IMG_W, IMG_H, 1)),
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU,
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU
        tf.keras.layers.Conv2D(filters=8, kernel_size=3,
activation='relu'),  # Conv-3×3×8, ReLU
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.Conv2D(filters=16, kernel_size=3,
activation='relu'),  # Conv-3×3×16, Relu,
        tf.keras.layers.MaxPool2D((2, 2)),  # MaxPool-2×2
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(units=5, activation='softmax',
use_bias='false')  # prediction layer
    ])
    print("--------------CNN_5 MODEL--------------")
    print(model_cnn_5.summary())

    if loss == None:
        # create binary cross entropy loss (one-hot encoding case)
        loss = tf.keras.losses.CategoricalCrossentropy()
    if optimizer == None:
        # create optimizer
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)

    # compile model for training
    model_cnn_5.compile(optimizer=optimizer, loss=loss,
metrics=["accuracy"])

    return model_cnn_5


# create model by name
def create_model(model_name, learning_rate):
    if model_name == "mlp_1":
        return create_mlp_1(learning_rate)
    elif model_name == "mlp_2":
        return create_mlp_2(learning_rate)
    elif model_name == "cnn_3":
        return create_cnn_3(learning_rate)
    elif model_name == "cnn_4":
        return create_cnn_4(learning_rate)
    elif model_name == "cnn_5":
        return create_cnn_5(learning_rate)
```

```
    else:
        return None


# save and load functions
def save_obj(obj, name):
    with open('part4/results/part4_' + name + '.pkl', 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def load_obj(name):
    with open('part4/results/part4_' + name + '.pkl', 'rb') as f:
        return pickle.load(f)


"""*************************************************************************
*******************************************"""
# my fit function
"""*************************************************************************
*******************************************"""
def my_fit(model_name, x_train, y_train, x_val, y_val, epoches=20):
    # declare and initialize some parameters
    split_size = len(y_train) // BS
    split_size_val = len(y_val) // BS
    length = x_train.shape[0]
    losses = []
    val_accs = []
    val_xb = np.array_split(x_val, split_size_val)
    val_yb = np.array_split(y_val, split_size_val)
    for i in LR:
        # in each iteration, create a new model
        model = create_model(model_name, i)
        # at each iteration, we have seperate loss and accuracy curves
        loss = []
        train_acc = []
        val_acc = []
        print("------------", model_name, " LR: ", i, "------------")
        for j in range(epoches):
            print("epoch: " + str(j) + " ----------------------------> ")

            # suffle the training set
            idxs = np.arange(0, length)
            np.random.shuffle(idxs)

            x_train = x_train[idxs]
            y_train = y_train[idxs]

            # extract batches
            train_xb = np.array_split(x_train, split_size)
            train_yb = np.array_split(y_train, split_size)

            # train the batches
            index = list(range(split_size))
            for i in index:
                results = model.train_on_batch(train_xb[i], train_yb[i])
                if i % 10 == 0:  # every 10 steps
                    # record the loss
                    loss.append(results[0])
                    # record the validaditon accuracy
                    rnd_idx = randrange(split_size_val)
```

```
                    acc_val = model.evaluate(val_xb[rnd_idx],
val_yb[rnd_idx])[1]
                    val_acc.append(acc_val)

            # record the loss and accuracy curves of each trial
            losses.append(np.array(loss))
            val_accs.append(np.array(val_acc))

    # convert list of np arrays to np arrays
    losses = np.array(losses)
    val_accs = np.array(val_accs)

    dic = {
        "name": model_name,
        "loss_curve_1": losses[0],
        "loss_curve_01": losses[1],
        "loss_curve_001": losses[2],
        "val_acc_curve_1": val_accs[0],
        "val_acc_curve_01": val_accs[1],
        "val_acc_curve_001": val_accs[2],
    }

    return dic


# my favorite architecture is cnn_5
dic = my_fit("cnn_5", train_img_conv, train_lbl, val_img_conv, val_lbl,
20) # 20
save_obj(dic, "cnn_5")

results = [dic]
part4Plots(results, save_dir='part4/plots', filename='part4_plot_1',
show_plot=True)


"""*************************************************************************
*******************************************"""
# Now, I will try to make scheduled learning rate to improve SGD based
training
"""*************************************************************************
*******************************************"""
def my_fit_2(model_name, x_train, y_train, x_val, y_val, LR=0.1,
epoches=30):
    # declare and initialize some parameters
    split_size = len(y_train) // BS
    split_size_val = len(y_val) // BS
    length = x_train.shape[0]
    val_accs = []
    val_xb = np.array_split(x_val, split_size_val)
    val_yb = np.array_split(y_val, split_size_val)

    # create a new model cnn_5 model
    optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)
    loss = tf.keras.losses.CategoricalCrossentropy()
    model = create_cnn_5(LR, loss, optimizer)

    # at each iteration, we have seperate loss and accuracy curves
    loss = []
    val_acc = []
    print("------------", model_name, "------------")
    for j in range(epoches):
```

```
        if j == 6:
            # create optimizer, change the learning rate
            optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
            loss = tf.keras.losses.CategoricalCrossentropy()
            # recompile model for training
            model.compile(optimizer=optimizer, loss=loss,
metrics=["accuracy"])

        print("epoch: " + str(j) + " -----------------------------> ")

        # suffle the training set
        idxs = np.arange(0, length)
        np.random.shuffle(idxs)

        x_train = x_train[idxs]
        y_train = y_train[idxs]

        # extract batches
        train_xb = np.array_split(x_train, split_size)
        train_yb = np.array_split(y_train, split_size)

        # train the batches
        index = list(range(split_size))
        for i in index:
            results = model.train_on_batch(train_xb[i], train_yb[i])
            if i % 10 == 0:  # every 10 steps
                # record the validation accuracy
                rnd_idx = randrange(split_size_val)
                acc_val = model.evaluate(val_xb[rnd_idx],
val_yb[rnd_idx])[1]
                val_acc.append(acc_val)

    # record the accuracy curves of each trial
    val_accs.append(np.array(val_acc))

    # convert list of np arrays to np arrays
    val_accs = np.array(val_accs)

    return val_accs


val_accs = my_fit_2("cnn_5", train_img_conv, train_lbl, val_img_conv,
val_lbl, 0.1, 30) # 30

dic2 = {
    "name": "cnn_5",
    "val_accs_rshp": np.mean(val_accs.reshape(-1, 10), axis=1),
    "val_accs": val_accs,
}

save_obj(dic2, "cnn_5_p2")


"""************************************************************************
*******************************************"""
# Repeat 2 and 3; however, in 3, continue training with 0.01 until the
epoch step that you determined
# in 5. Then, set the learning rate to 0.001 and continue training until
30 epochs.
"""************************************************************************
*******************************************"""
```

```
def my_fit_3(model_name, x_train, y_train, x_val, y_val, LR=0.1, 
epoches=30):
    # declare and initialize some parameters
    split_size = len(y_train) // BS
    split_size_val = len(y_val) // BS
    length = x_train.shape[0]
    val_accs = []
    val_xb = np.array_split(x_val, split_size_val)
    val_yb = np.array_split(y_val, split_size_val)

    # create a new model cnn_5 model
    optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)
    loss = tf.keras.losses.CategoricalCrossentropy()
    model = create_cnn_5(LR, loss, optimizer)

    # at each iteration, we have seperate loss and accuracy curves
    loss = []
    val_acc = []
    print("------------", model_name, "------------")
    for j in range(epoches):
        if (j == 6):
            # create optimizer, change the learning rate
            optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
            loss = tf.keras.losses.CategoricalCrossentropy()
            # recompile model for training
            model.compile(optimizer=optimizer, loss=loss, 
metrics=["accuracy"])
        elif (j == 10):
            # create optimizer, change the learning rate
            optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
            loss = tf.keras.losses.CategoricalCrossentropy()
            # recompile model for training
            model.compile(optimizer=optimizer, loss=loss, 
metrics=["accuracy"])

        print("epoch: " + str(j) + " -----------------------------> ")

        # suffle the training set
        idxs = np.arange(0, length)
        np.random.shuffle(idxs)

        x_train = x_train[idxs]
        y_train = y_train[idxs]

        # extract batches
        train_xb = np.array_split(x_train, split_size)
        train_yb = np.array_split(y_train, split_size)

        # train the batches
        index = list(range(split_size))
        for i in index:
            results = model.train_on_batch(train_xb[i], train_yb[i])
            if i % 10 == 0:  # every 10 steps
                # record the validaditon accuracy
                rnd_idx = randrange(split_size_val)
                acc_val = model.evaluate(val_xb[rnd_idx], 
val_yb[rnd_idx])[1]
                val_acc.append(acc_val)

    # record theaccuracy curves of each trial
    val_accs.append(np.array(val_acc))
```

```
    # convert list of np arrays to np arrays
    val_accs = np.array(val_accs)

    return val_accs


val_accs = my_fit_3("cnn_5", train_img_conv, train_lbl, val_img_conv,
val_lbl, 0.1, 30)  # 30

dic3 = {
    "name": "cnn_5_schl_3",
    "val_accs_rshp": np.mean(val_accs.reshape(-1, 10), axis=1),
    "val_accs": val_accs,
}

save_obj(dic3, "cnn_5_p3")

dic1 = load_obj("cnn_5")
dic2 = load_obj("cnn_5_p2")
dic3 = load_obj("cnn_5_p3")

curves = {
    "name": "cnn_5_scheduled_0.01",
    "loss_curve_1": (np.array([])),
    "loss_curve_01": (np.array([])),
    "loss_curve_001": (np.array([])),
    "val_acc_curve_1": (np.mean(dic1["val_acc_curve_1"].reshape(-1, 10),
axis=1))[0:30],
    # np.mean(dic5["val_acc_curve_1"].reshape(-1, 10), axis=1),
    "val_acc_curve_01": (dic2["val_accs_rshp"])[0:30],
    "val_acc_curve_001": (dic3["val_accs_rshp"])[0:30],
}  # dic["val_accs_rshp"],

results = [curves]
part4Plots(results, save_dir='part4/plots', filename='part4_plot_2',
show_plot=True)
```