EE449 HW1 NEURAL NETWORKS

Kutay Uğurlu 2232841

July 31, 2021

1 Basic Concepts

1.1 Which Function?

1. The networks approximate the function defining a hyperplane $F : R^{784} \mapsto R$ in 784 dimensional space seperating the whole space into 5 regions.

2. The loss function categorical cross entropy $\mathcal{L}$ is defined in a way that it penalizes every unmatching prediction with the ground truth label: $\mathcal{L} = -\frac{1}{\#batches} \sum_{batches} [i_{true} \log(i_{pred}) + (1 - i_{true}) \log(1 - i_{pred})]$. For matching prediction and label pairs the $\mathcal{L}$ outputs 0, on the other hand for unmatching pairs, one of the terms results in 0 and the ramining results in negative number large in magnitude, which results in high loss with the sign inversion by the minus in the loss function.

1.2 Gradient Computation

The weight update rule in back propagation step of Gradient Descent algorithm is $w_{k+1} = w_k - \gamma \nabla_w \mathcal{L}\Big|_{w=w_k}$. Hence, $\nabla_w \mathcal{L}\Big|_{w=w_k}$ can be rewritten as $\frac{w_k - w_{k+1}}{\gamma}$.

1.3 Some Training Parameters and Basic Parameter Calculation

1. **Epoch** is the number of passes of entire training dataset to the model. **Batch** is a subset of data, *i.e.*, the training data partitioned into smaller samples.

2. The number of batches of size B is $\lfloor \frac{N}{B} \rfloor$, and there is one extra "semi-batch". This leads to a number of $\lceil \frac{N}{B} \rceil$ batches.

3. The number of SGD iterations for E epochs with a training set with N samples and batch size of B is $E.\lceil \frac{N}{B} \rceil$

1.4 Computing Number of Parameters of ANN Classifier

1. In an MLP, the number of parameters in a layer is the product of hidden units in the current layer, H_k, times the number of hidden units in the previous layer, H_{k-1}. Using this, the number of parameters in an MLP can be written as $H_1 D_{in} + H_k D_{out} + \sum_{k=2}^{K} H_k H_{k-1} + \sum_{k=1}^{K} H_k$,where H_k represents also the number of bias paramters in the k^{th} layer,in the compact form.

2. In a Convolutional Neural Network, the number of parameters in a convolutional layer is $(KernelSize) \times (\#Kernels)$. Hence the number of parameters in the given CNN is $\sum_{k=1}^{K} H_k W_k C_k$.

2 Experimenting ANN Architectures

2.1 Training Experiment Plots

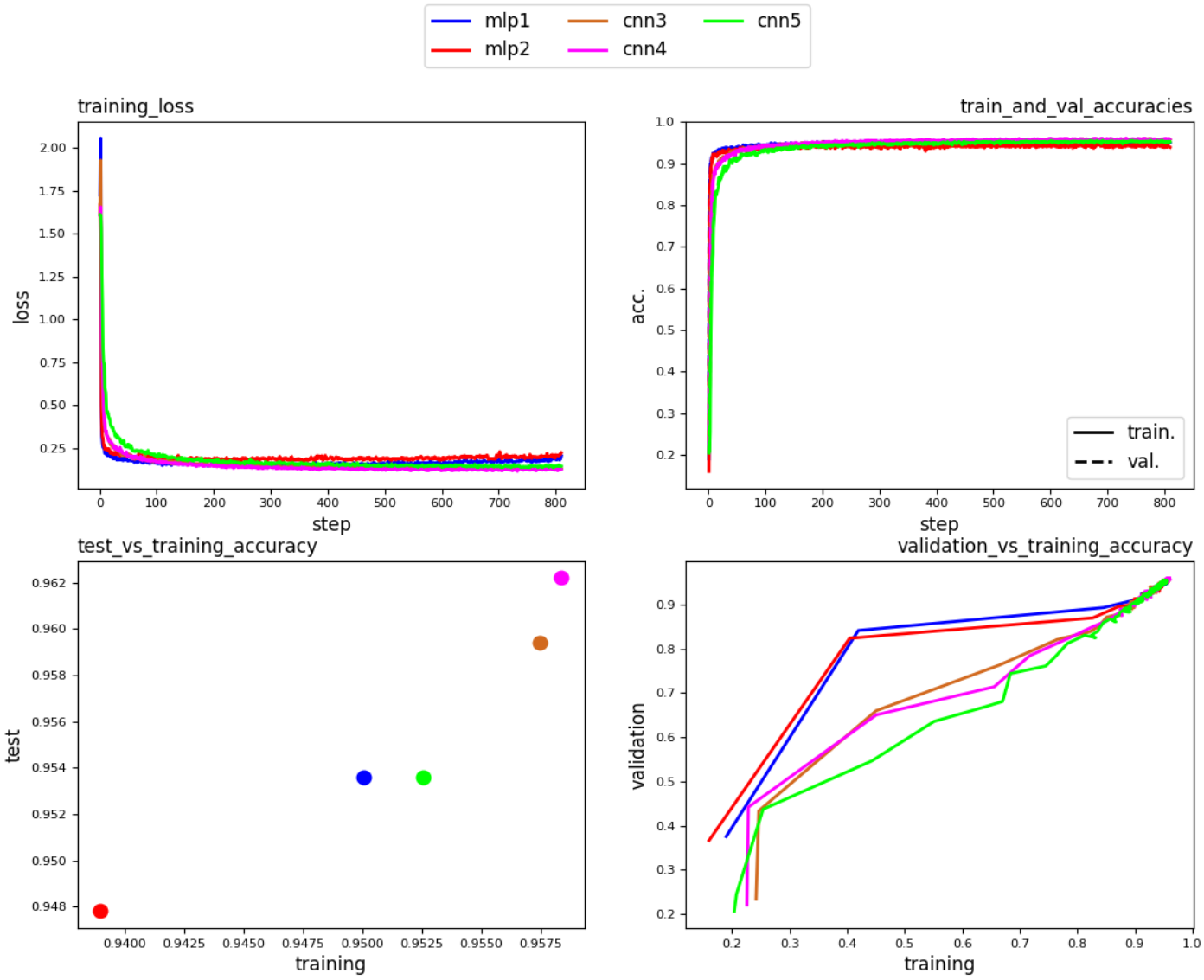
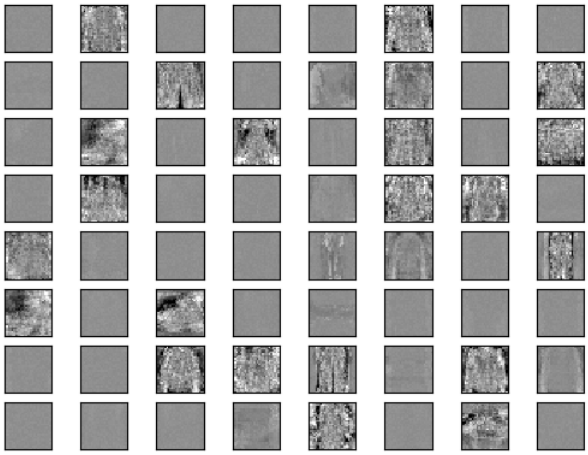


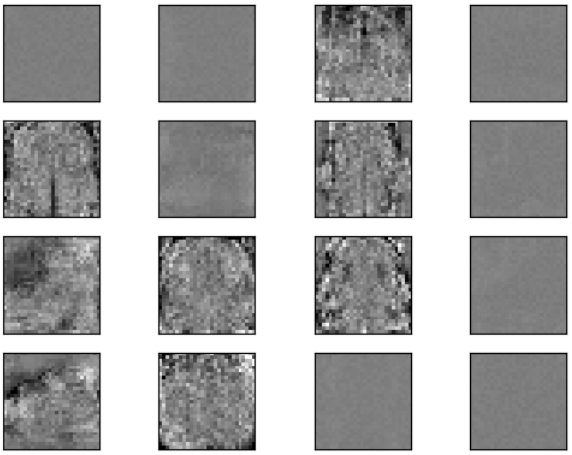
Figure 1: Performance Plots for Different Models

2.2 Weight Visualizations

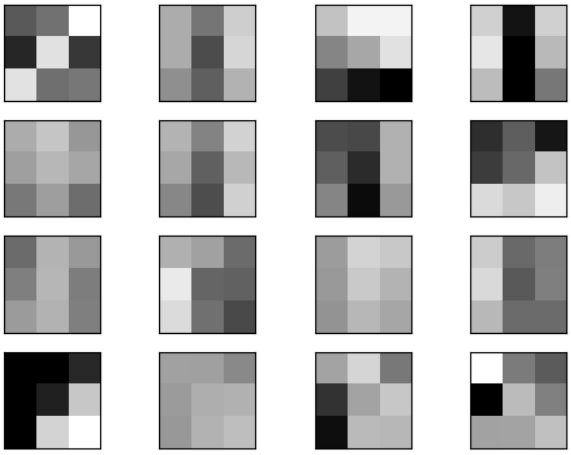
The magnitude of the weights of the first layers regarding each architecture are shown below.



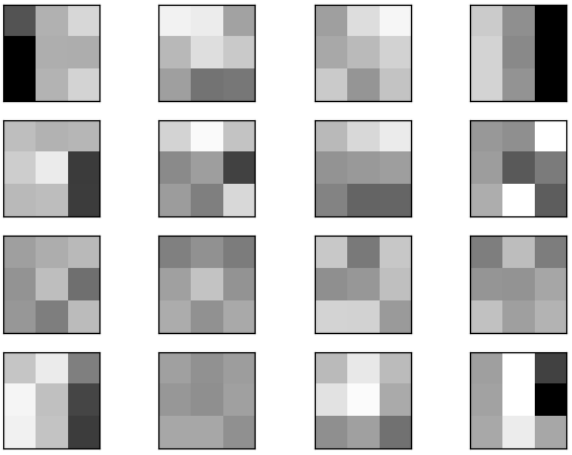
(a) mlp1



(b) mlp2



(c) cnn3



(d) cnn4

Figure 2: Weight Visualization of architectures

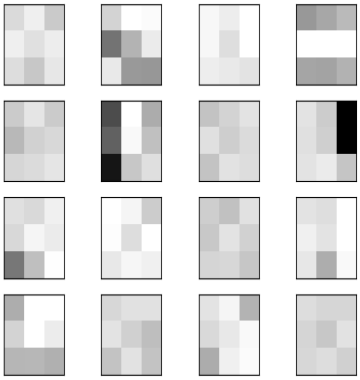


Figure 3: Weights Visualizations of cnn5

2.3 Discussions

1. The generalization performance of a classifier is the performance of the model with response to "previously unseen data", *i.e.*, test dataset. Hence, the generalization performance of a classifier is the test accuracy.

2. In this sense, the test accuracy plots inform about the generalization performance of the classifier networks.

3. MultiLayer Perceptrons are more successful at classifiying the images in terms of test accuracy, although the difference is not much significant. The difference in the test accuracies of MLP models may be attributed to either the reduced non-linearity of the mlp2 architecture or the increased number of parameters.

4. The number of parameters are higher in the mlp2 model, it had higher test accuracy. For the convolutional neural networks, the models cnn3, cnn4 and cnn5 have 9741, 7253 and 6061 number of trainable parameters, respectively. For the overall architecture type, MLPs, having more parameters, resulted in better training and test accuracy. However, the features of CNN's such as the kernel weights "scanning" over the image provides a chance for them to achieve a test accuracy not significantly less than MLPS, although having 5-10 times less parameters than mlp2 model.

5. For the models sharing the same type of architectures, it can be observed that the higher the number of layers, *i.e.* the deeper the network, the higher training accuracy achieved. For the test accuracy, on the other hand, the same relation holds for MLPs, but no specific correlation was observed in test accuracies of CNN with the depth of the networks. Usually, the deeper networks are expected to learn more complicated functions, hence provides a more accurate seperation of input space.

6. Yes, they show the magnitude of the weight associated to the relevant input, implying the "importance" the model puts on the matching pixel.

7. Yes, some units of the first layer of the MLPs seem to specialize to specific classes. *E.g.* the visualization at the last column of first row of Figure 2b, showing the 28x28 weights reshaped from 1x784 weight vector that the relevant unit learned, seems to have specialized to class "bottom-wear".

8. The weights of MLPs are more interpretable. Since they do not share the weights on the whole input, the weights can be associated to specific pixel of the input. This is why, the observation in 7 was possible to make.

9. The MultiLayer Perceptrons are designed to take flattened inputs, whereas CNNs are designed to take inputs as images. Since CNN filters scan through the whole input and hidden layer activations, they do not require a weight assigned specifically to a correspondant pixel, resulting in less number of parameters. Although having less parameters, they show almost similar success at classifying images, since the activation of the CNN also contain the information about neighboring pixels, due to the nature of 2-dimensional convolution operation.

10. I would choose mlp2 to be used in the future, due to its generalization performance compared to others.

3 Experimenting Activation Functions

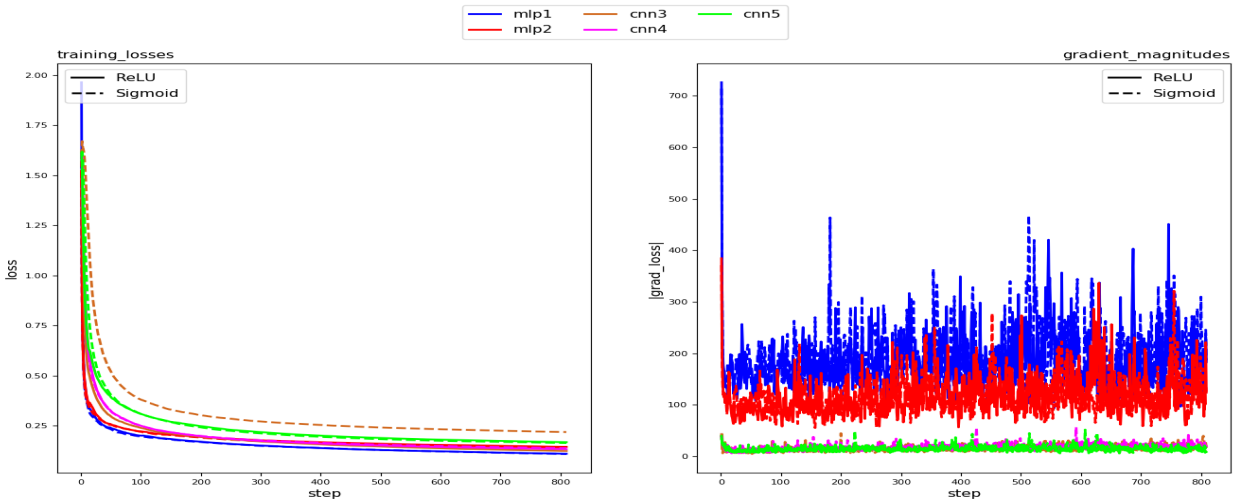


Figure 4: Comparison of Different Activation Functions

1. As it can be seen on the subplot on the right of Figure 4, CNNs have significantly lower magnitude of gradient loss. Also, excluding cnn3 model from comparison due to its lacking activation - *I have made an experiment with mlp2 model with/without activation and observed that it matters* -, it is observed that cnn4 has higher gradient loss than cnn5. One may conclude that increasing depth results in lower gradient loss in the first layer.

2. The gradient difference between different architectures is most probably due to the number of parameters in the first layer, MLPs whose first layer having significantly higher number of parameters, resulted in higher Frobenius norm. Furthermore, for both mlp1-mlp2 and cnn4-cnn5 pairs it is observed that the deeper network had lower magnitude of gradient loss. This is de to the calculation of gradient using Chain Rule. Sigmoid function's derivative takes values in range [0,1], in the chain rule expression these partial derivatives are multiplied and results in smaller gradient in every layer further from the last layer. Similarly ReLU's derivative takes values 0 and 1. Even if it is faster to compute, using ReLU causes no contribution to learning when te unit is not active, resulting in 0 gradient. In larger networks, it becomes more important since it may cause the networ to stop learning in the worst case. This problem is known as **Vanishing Gradients Problem**.

3. $$f(a) = sigmoid(a) = \frac{1}{1 - e^{a}} \ , \ \frac{\partial f(a)}{\partial a} = f(a)[1 - f(a)] \tag{1}$$

   The value of sigmoid's derivative approaches to zero out of the range [-1,1]. Evaluating derivative at higher points will result in smaller gradients and the model will end up learning significantly slower, it will require more epochs to be trained to achieve the same training accuracy. Normalizing the inputs between [-1,1] range will scale the input to be in the linear portion of the sigmoid, hence the weight update procedure will be more efficient.

4 Experimenting Learning Rate

4.1 Experimenting Different Learning Rates without Scheduling

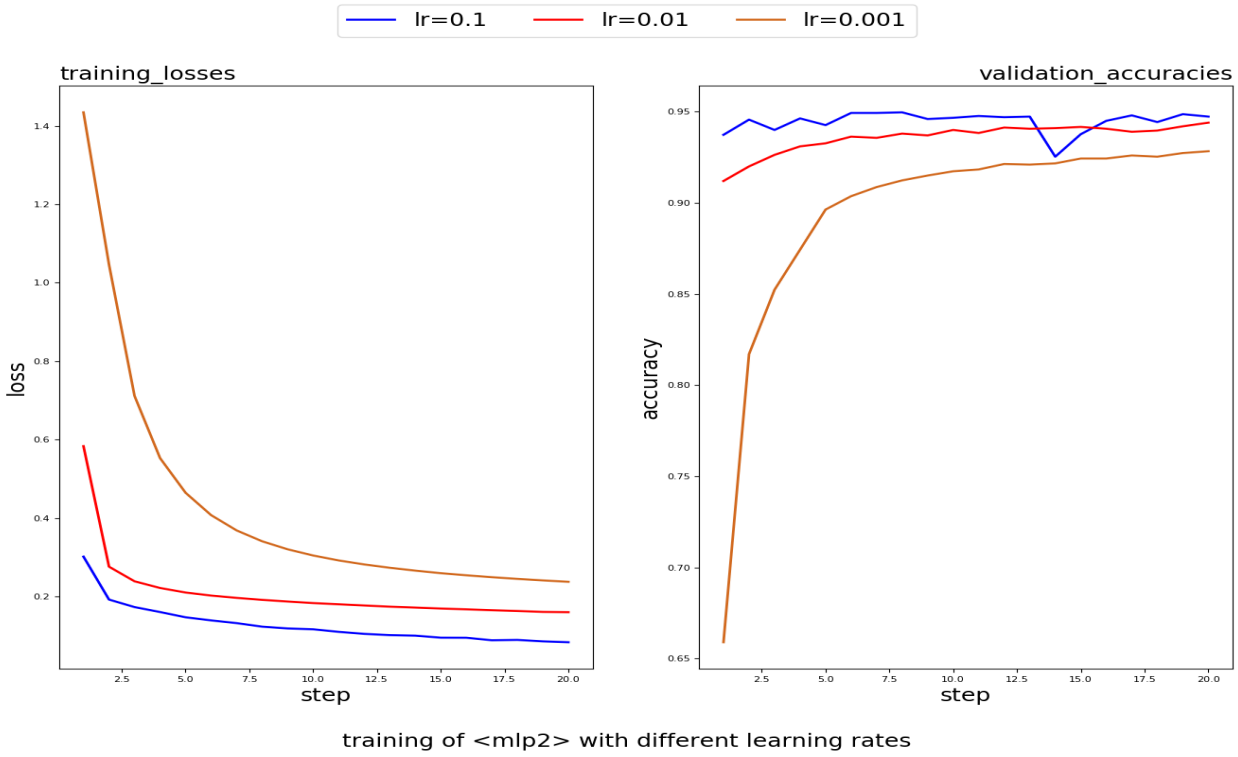
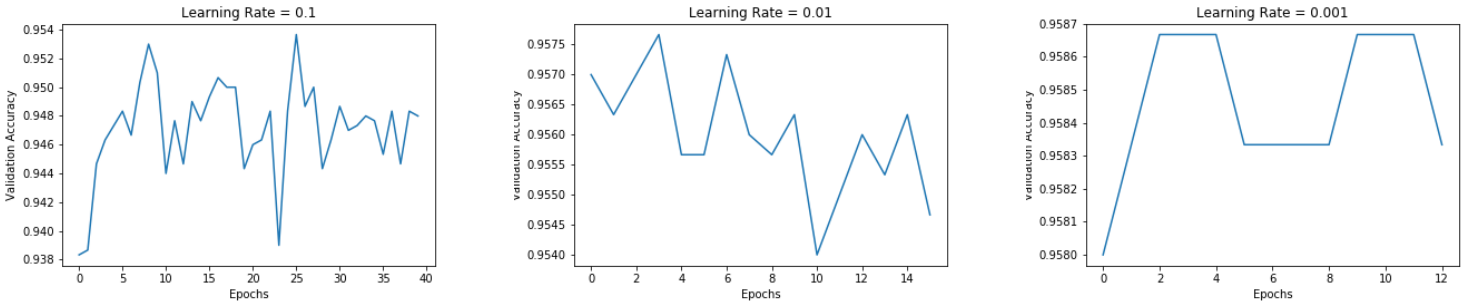


Figure 5: Performance Plots for Different Learning Rates

4.2 Experimenting Different Learning Rates with Scheduling



(a) Learning Rate = 0.1 (b) Learning Rate = 0.01 (c) Learning Rate = 0.0001

Figure 6: Validation Accuracy curve for different learning rates

4.3 Discussions

1. By looking at Figure 5, it can be concluded that the larger learning rate values resulted in faster convergence. This is expected since parameters are updated in larger magnitude of values, so the final values are achieved faster.

2. For both loss and accuracy plots, smaller learning rate values resulted in smoother curves. Although the blue curve corresponding to $\gamma = 0.1$ fluctuates around 0.95, brown curve ($\gamma = 0.001$) seems to converge some value around 0.92.

3. Yes, the scheduled learning rate approach worked. It can be clearly seen from downscaling validation accuracy axis of Figure 6 that the validation accuracy converged better. In addition to this, compared to the model in 2.1, the test accuracy was calculated to be almost as same, although trained in a significantly less amount of time.

4. With the constant learning rate of 0.01, the convergence of validation accuracy curve of Adam in Part 2.1 was relatively smooth. The test accuracy comparison was made in the previous part. Convergence speed was significantly higher in the scheduled learning rate experiment, making it a more favorable option, considering also that the architecture in Part 2.1, was selected as the one having the best test accuracy among 10 training experiments, hence 10 different models.

The scheduled learning experiment was conducted in Jupyter notebook, and the procedure can be seen in the notebook output in the Appendix section.

5 APPENDIX

5.1 Part 2 Training

```
import numpy as np
import tensorflow as tf
import json
import h5py

from tensorflow.keras.layers import Dense as Dense
from tensorflow.keras.layers import Conv2D as Conv2D
from tensorflow.keras.layers import Input as Input
from tensorflow.keras.layers import MaxPooling2D as MaxPooling2D
from tensorflow.keras.layers import GlobalAveragePooling2D as GlobalAvgPooling2D
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split

# HYPERPARAMETERS
BS = 50
EPOCHS = 15

# MODELS
```

```
def create_mlp1():
    mlp_1 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=64, activation="relu"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_1


def create_mlp2():
    mlp_2 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=16, activation="relu"),
            Dense(units=64, use_bias=False, activation=None),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_2


# strides = (1,1) and valid padding is already default for both pooling and conv2d.


def create_cnn3():
    cnn_3 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(7, 7), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(5, 5), activation=None),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_3


def create_cnn4():
    cnn_4 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(5, 5), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(5, 5), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
```

```
        ]
    )
    return cnn_4


def create_cnn5():
    cnn_5 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_5


## DATA
train_labels = np.load("dataset\\train_labels.npy")
test_labels = np.load("dataset\\test_labels.npy")
train_images = np.load("dataset\\train_images.npy")
test_images = np.load("dataset\\test_images.npy")


## [-1, 1] Scaling
############################################################
test_images = (
    2 * (test_images - test_images.min()) / (test_images.max() - test_images.min())
)  # -1 to 1 scaling
############################################################
train_images = (
    2 * (train_images - train_images.min()) / (train_images.max() - train_images.min())
)  # -1 to 1 scaling
############################################################


train_image1D, validate_image1D, train_label1D, validate_label1D = train_test_split(
    train_images, train_labels, test_size=0.1, random_state=42
)

## CNN's are going to take input as images. So here is the reshaping:
(samples, rectified) = train_images.shape
train_images_2D = np.reshape(train_images, newshape=(samples, 28, 28, 1))
(samples, rectified) = test_images.shape
test_images_2D = np.reshape(test_images, newshape=(samples, 28, 28, 1))

train_image2D, validate_image2D, train_label2D, validate_label2D = train_test_split(
```

```
    train_images_2D, train_labels, test_size=0.1, random_state=42
)

(samples, rectified) = train_image1D.shape
ITERATION_PER_EPOCH = np.floor(samples // BS)

RESULTS_DICT = {}

for i in range(10):

    model = create_cnn3()
    model_name = "cnn3"

    ## Model parameters
    optimizer = Adam(learning_rate=0.01)
    loss = tf.keras.losses.SparseCategoricalCrossentropy()
    metric = tf.keras.metrics.SparseCategoricalAccuracy()
    model.compile(optimizer=optimizer, loss=loss, metrics=[metric])
    print("MODEL COMPILED")

    ## Containers
    training_acc_curve = []
    training_loss_curve = []
    validation_acc_curve = []

    for epoch in range(int(EPOCHS)):
        for batch in range(int(ITERATION_PER_EPOCH)):

            train_data = train_image2D[
                batch * BS : (batch + 1) * BS, :, :, :
            ]  # Drop last 2 dimension for training MLPs
            train_label = train_label2D[batch * BS : (batch + 1) * BS]
            train_dict = model.train_on_batch(
                x=train_data, y=train_label, reset_metrics=False, return_dict=True
            )
            # If current iteration is multiple of 10, get training metrics
            if (epoch * BS + batch) % 10 == 0:
                train_loss, train_acc = (
                    train_dict["loss"],
                    train_dict["sparse_categorical_accuracy"],
                )
                training_acc_curve.append(train_acc)
                training_loss_curve.append(train_loss)
                val_loss, val_acc = model.evaluate(
                    x=validate_image2D, y=validate_label2D, batch_size=BS
                )
                validation_acc_curve.append(val_acc)

        # Create Permutation
        permute_idx = np.random.permutation(samples)

        ## Shuffle Training data for MLP
        train_image1D = train_image1D[permute_idx, :]
```

```
            ## Shuffle Training data for CNN
            train_image2D = train_image2D[permute_idx, :, :, :]

            ## Shuffle Labels
            train_label1D = train_label1D[permute_idx]
            train_label2D = train_label2D[permute_idx]

        ## Get the weights of the first trainable layer
        first_layer = model.trainable_weights[0].numpy().tolist()

        ## Get test accuracy
        test_loss, test_acc = model.evaluate(x=test_images_2D, y=test_labels, batch_size=BS
                                                    )

        ## Create dictionary
        result_dict = {}
        result_dict["train_acc_curve"] = training_acc_curve
        result_dict["train_loss_curve"] = training_loss_curve
        result_dict["validation_acc_curve"] = validation_acc_curve
        result_dict["first_layer"] = first_layer
        result_dict["test_acc"] = test_acc

        RESULTS_DICT["Trial" + str(i + 1)] = result_dict


    ## Save dictionary
    with open("PART2RESULTS/" + "cnn5" + "_results.json", "w") as fp:
        json.dump(RESULTS_DICT, fp)

    ## Save model
    print("[INFO] Loop ended. Saving model.")
    model.save(model_name + ".model")
```

5.2 Part 2 Results

```
    import json
import matplotlib.pyplot as plt
from utils import part2Plots
from utils import visualizeWeights
import numpy as np

model_names = ["mlp1", "mlp2", "cnn3", "cnn4", "cnn5"]

for model_name in model_names:

    file = "PART2RESULTS\\new\\" + model_name + "_results.json"
    with open(file) as json_file:
        data = json.load(json_file)

    train_acc = np.zeros((1, 810))
    val_acc = np.zeros((1, 810))
    train_loss = np.zeros((1, 810))
```

```
    for item in list(data.keys()):
        res_dict = data[item]
        train_acc += np.array(res_dict["train_acc_curve"])
        train_loss += res_dict["train_loss_curve"]
        val_acc += res_dict["validation_acc_curve"]

    test_accuracies = [data[item]["test_acc"] for item in data.keys()]
    max_test_acc = max(test_accuracies)
    maximum_test_acc_model = "Trial" + str(test_accuracies.index(max_test_acc) + 1)

    first_layer = data[maximum_test_acc_model]["first_layer"]
    visualizeWeights(
        np.array(first_layer),
        save_dir="PART2RESULTS\\weights",
        filename=model_name + "weights",
    )
    print("SHAPES: ", np.array(first_layer).shape)

    mean_train_acc = (train_acc * 0.1).tolist()[0]
    mean_train_loss = (train_loss * 0.1).tolist()[0]
    mean_val_acc = (val_acc * 0.1).tolist()[0]

    RESULTS_DICT = {}
    RESULTS_DICT["test_acc"] = max_test_acc
    RESULTS_DICT["weights"] = first_layer
    RESULTS_DICT["train_acc_curve"] = mean_train_acc
    RESULTS_DICT["val_acc_curve"] = mean_val_acc
    RESULTS_DICT["loss_curve"] = mean_train_loss
    RESULTS_DICT["name"] = model_name

    ## Save dictionary
    with open("PART2RESULTS/Part2_" + model_name + "_final_results.json", "w") as fp:
        json.dump(RESULTS_DICT, fp)

ALL_DICTS = []
for model_name in model_names:
    file = "PART2RESULTS\\Part2_" + model_name + "_final_results.json"
    with open(file) as json_file:
        data = json.load(json_file)
    ALL_DICTS.append(data)

part2Plots(
    ALL_DICTS, save_dir="PLOTS", filename="Part2_AllPlots"
)


with open(file) as json_file:
    data = json.load(json_file)
```

5.3 Part 3 Training

```
    import numpy as np
import tensorflow as tf
import json
import h5py

from tensorflow.keras.layers import Dense as Dense
from tensorflow.keras.layers import Conv2D as Conv2D
from tensorflow.keras.layers import Input as Input
from tensorflow.keras.layers import MaxPooling2D as MaxPooling2D
from tensorflow.keras.layers import GlobalAveragePooling2D as GlobalAvgPooling2D
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split

# HYPERPARAMETERS
BS = 50
EPOCHS = 15

# MODELS


def create_mlp1():
    mlp_1 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=64, activation="relu"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_1


def create_mlp2():
    mlp_2 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=16, activation="relu"),
            Dense(units=64, use_bias=False, activation=None),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_2


# strides = (1,1) and valid padding is already default for both pooling and conv2d.


def create_cnn3():
    cnn_3 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(7, 7), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(5, 5), activation=None),
```

```
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_3


def create_cnn4():
    cnn_4 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(5, 5), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(5, 5), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_4


def create_cnn5():
    cnn_5 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_5


def create_mlp1_sig():
    mlp_1 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=64, activation="relu"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_1
```

```
def create_mlp2_sig():
    mlp_2 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=16, activation="relu"),
            Dense(units=64, use_bias=False, activation=None),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_2


def create_cnn3_sig():
    cnn_3 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(7, 7), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(5, 5), activation=None),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_3


def create_cnn4_sig():
    cnn_4 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(5, 5), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(5, 5), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_4


def create_cnn5_sig():
    cnn_5 = tf.keras.Sequential(
        [
            Input(shape=(28, 28, 1)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=8, kernel_size=(3, 3), activation="relu"),
```

```
            MaxPooling2D(pool_size=(2, 2)),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            Conv2D(filters=16, kernel_size=(3, 3), activation="relu"),
            MaxPooling2D(pool_size=(2, 2)),
            GlobalAvgPooling2D(data_format="channels_last"),
            Dense(units=5, activation="softmax"),
        ]
    )
    return cnn_5


## DATA
train_labels = np.load("dataset\\train_labels.npy")
test_labels = np.load("dataset\\test_labels.npy")
train_images = np.load("dataset\\train_images.npy")
test_images = np.load("dataset\\test_images.npy")


## [-1, 1] Scaling
#########################################################
test_images = (
    2 * (test_images - test_images.min()) / (test_images.max() - test_images.min())
)  # -1 to 1 scaling
#########################################################
train_images = (
    2 * (train_images - train_images.min()) / (train_images.max() - train_images.min())
)  # -1 to 1 scaling
#########################################################


train_image1D, validate_image1D, train_label1D, validate_label1D = train_test_split(
    train_images, train_labels, test_size=0.1, random_state=42
)

## CNN's are going to take input as images. So here is the reshaping:


(samples, rectified) = test_images.shape
test_images_2D = np.reshape(test_images, newshape=(samples, 28, 28, 1))
(samples, rectified) = train_images.shape
train_images_2D = np.reshape(train_images, newshape=(samples, 28, 28, 1))


train_image2D, validate_image2D, train_label2D, validate_label2D = train_test_split(
    train_images_2D, train_labels, test_size=0.1, random_state=42
)

(samples, rectified) = train_image1D.shape
print(samples)

ITERATION_PER_EPOCH = np.floor(samples // BS)

## Containers
relu_loss_curve = []
```

```
sigmoid_loss_curve = []
relu_layers = []
sigmoid_layers = []
RESULTS_DICT = {}

model = create_cnn5()

## Model parameters
optimizer = Adam(learning_rate=0.01)
loss = tf.keras.losses.SparseCategoricalCrossentropy()
metric = tf.keras.metrics.SparseCategoricalAccuracy()
model.compile(optimizer=optimizer, loss=loss, metrics=[metric])


for epoch in range(int(EPOCHS)):
    for batch in range(int(ITERATION_PER_EPOCH)):

        train_data = train_image2D[
            batch * BS : (batch + 1) * BS, :, :, :
        ]  # Drop last 2 dimension for training MLPs
        train_label = train_label2D[batch * BS : (batch + 1) * BS]
        train_dict = model.train_on_batch(
            x=train_data, y=train_label, reset_metrics=False, return_dict=True
        )
        # If current iteration is multiple of 10, get training metrics
        if (epoch * BS + batch) % 10 == 0:
            train_loss, train_acc = (
                train_dict["loss"],
                train_dict["sparse_categorical_accuracy"],
            )
            relu_loss_curve.append(train_loss)
            ## Get the weights of the first trainable layer
            first_layer = model.trainable_weights[0].numpy()
            relu_layers.append(first_layer)

    # Create Permutation
    permute_idx = np.random.permutation(samples)

    ## Shuffle Training data for MLP
    train_image1D = train_image1D[permute_idx, :]

    ## Shuffle Training data for CNN
    train_image2D = train_image2D[permute_idx, :, :, :]

    ## Shuffle Labels
    train_label1D = train_label1D[permute_idx]
    train_label2D = train_label2D[permute_idx]


model = create_cnn5_sig()
model_name = "cnn5"

## Model parameters
optimizer = Adam(learning_rate=0.01)
```

```
loss = tf.keras.losses.SparseCategoricalCrossentropy()
metric = tf.keras.metrics.SparseCategoricalAccuracy()
model.compile(optimizer=optimizer, loss=loss, metrics=[metric])


for epoch in range(int(EPOCHS)):
    for batch in range(int(ITERATION_PER_EPOCH)):

        train_data = train_image2D[
            batch * BS : (batch + 1) * BS, :, :, :
        ]  # Drop last 2 dimension for training MLPs
        train_label = train_label2D[batch * BS : (batch + 1) * BS]
        train_dict = model.train_on_batch(
            x=train_data, y=train_label, reset_metrics=False, return_dict=True
        )
        # If current iteration is multiple of 10, get training metrics
        if (epoch * BS + batch) % 10 == 0:
            train_loss, train_acc = (
                train_dict["loss"],
                train_dict["sparse_categorical_accuracy"],
            )
            sigmoid_loss_curve.append(train_loss)

            ## Get the weights of the first trainable layer
            first_layer = model.trainable_weights[0].numpy()
            sigmoid_layers.append(first_layer)

    # Create Permutation
    permute_idx = np.random.permutation(samples)

    ## Shuffle Training data for MLP
    train_image1D = train_image1D[permute_idx, :]

    ## Shuffle Training data for CNN
    train_image2D = train_image2D[permute_idx, :, :, :]

    ## Shuffle Labels
    train_label1D = train_label1D[permute_idx]
    train_label2D = train_label2D[permute_idx]

## Calculate loss gradient from first layer weights

sigmoid_grad_curve = [
    np.linalg.norm(sigmoid_layers[i] - sigmoid_layers[i + 1]) / 0.01
    for i in range(len(sigmoid_layers) - 1)
]
relu_grad_curve = [
    np.linalg.norm(relu_layers[i] - relu_layers[i + 1]) / 0.01
    for i in range(len(relu_layers) - 1)
]


## Create dictionary
result_dict = {}
```

```
result_dict["name"] = model_name
result_dict["relu_loss_curve"] = relu_loss_curve
result_dict["relu_grad_curve"] = relu_grad_curve
result_dict["sigmoid_loss_curve"] = sigmoid_loss_curve
result_dict["sigmoid_grad_curve"] = sigmoid_grad_curve

## Save dictionary
with open("PART3RESULTS/" + model_name + "new_results.json", "w") as fp:
    json.dump(result_dict, fp)
```

5.4 Part 3 Results

```
    import matplotlib.pyplot as plt
from utils import part3Plots
import json

model_names = ["mlp1", "mlp2", "cnn3", "cnn4", "cnn5"]
container = []

for model_name in model_names:
    file = "PART3RESULTS\\" + model_name + "_new_results.json"
    with open(file) as json_file:
        data = json.load(json_file)
    container.append(data)

part3Plots(
    container,
    save_dir="PART3RESULTS",
    filename="Part3_results",
)
```

EE449_HW1_Part_4_Scheduled_Learning

April 16, 2021

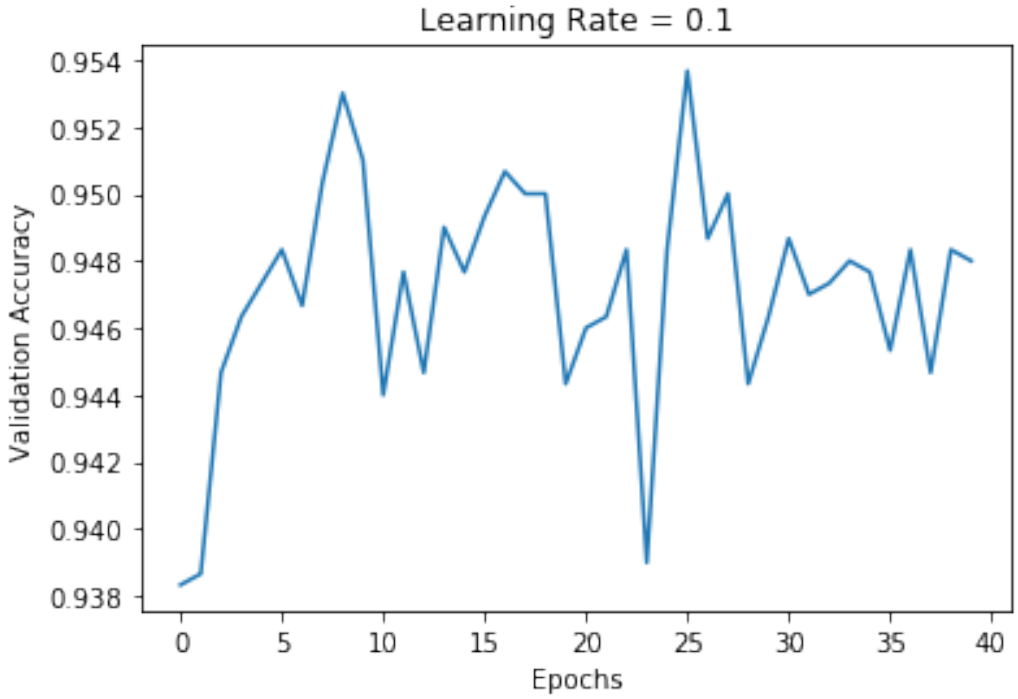
1 5.5 Part 4 Scheduled Learning

[1]:
```
import numpy as np
import tensorflow as tf
import json
from tensorflow.keras.layers import Dense as Dense
from tensorflow.keras.layers import Conv2D as Conv2D
from tensorflow.keras.layers import Input as Input
from tensorflow.keras.layers import MaxPooling2D as MaxPooling2D
from tensorflow.keras.layers import GlobalAveragePooling2D as GlobalAvgPooling2D
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from random import shuffle
# HYPERPARAMETERS
BS = 50
EPOCHS = 20
LR = 0.001
# MODEL
def create_mlp2():
    mlp_2 = tf.keras.Sequential(
        [
            Input(shape=(784,)),
            Dense(units=16, activation="relu"),
            Dense(units=64, use_bias=False, activation=None),
            Dense(units=5, activation="softmax"),
        ]
    )
    return mlp_2
# DATA
train_labels = np.load("dataset\\train_labels.npy")
test_labels = np.load("dataset\\test_labels.npy")
train_images = np.load("dataset\\train_images.npy")
test_images = np.load("dataset\\test_images.npy")
## [-1, 1] Scaling
############################################################
test_images = (
```

```
    2 * (test_images - test_images.min()) / (test_images.max() - test_images.
 ↪min())
)  # -1 to 1 scaling
############################################################
train_images = (
    2 * (train_images - train_images.min()) / (train_images.max() -␣
 ↪train_images.min())
)  # -1 to 1 scaling
############################################################
train_image1D, validate_image1D, train_label1D, validate_label1D =␣
 ↪train_test_split(
    train_images, train_labels, test_size=0.1, random_state=42
)
```

[7]:
```
model_name = "mlp2"
model = create_mlp2()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)
loss = tf.keras.losses.SparseCategoricalCrossentropy()
metrics = tf.keras.metrics.SparseCategoricalAccuracy()
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
H = model.fit(
    x=train_image1D,
    y=train_label1D,
    shuffle=True,
    batch_size=BS,
    epochs=24,
    validation_data=(validate_image1D, validate_label1D),
    verbose = 0
)
```

[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(H.history["val_sparse_categorical_accuracy"])
plt.xlabel('Epochs')
plt.ylabel("Validation Accuracy")
plt.title("Learning Rate = 0.1")
plt.savefig("SLR01")
```

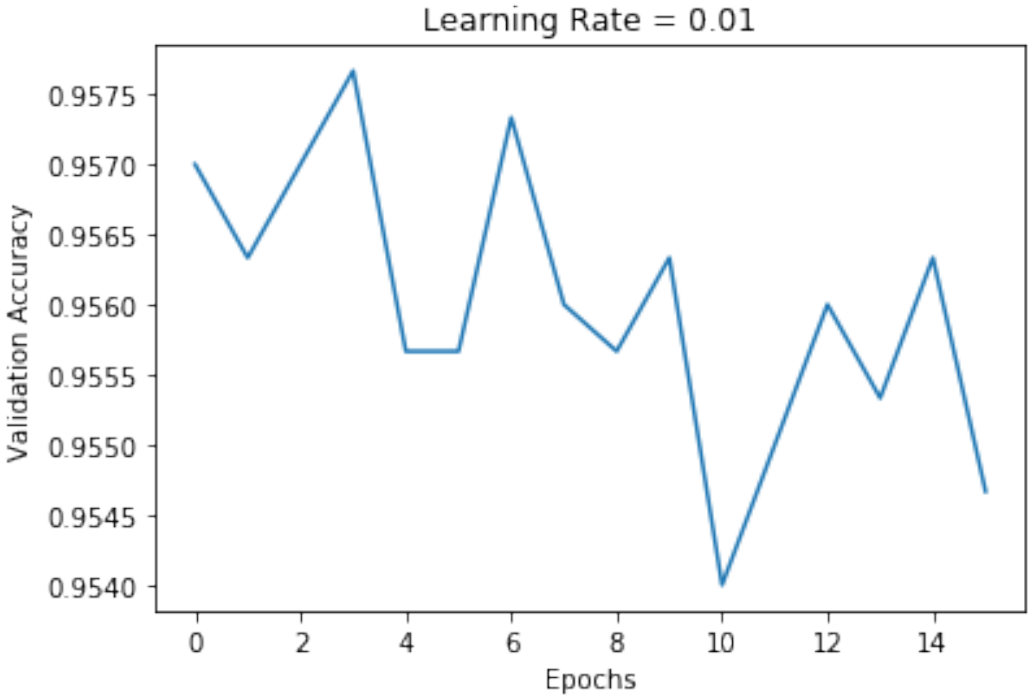


Accuracy stopped increasing at epoch 24. Training 16 epochs for learning_rate = 0.01

[8]:
```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

H2 = model.fit(
    x=train_image1D,
    y=train_label1D,
    shuffle=True,
    batch_size=BS,
    epochs=3,
    validation_data=(validate_image1D, validate_label1D),
    verbose=0
)
```

[6]:
```
plt.plot(H2.history["val_sparse_categorical_accuracy"])
plt.xlabel('Epochs')
plt.ylabel("Validation Accuracy")
plt.title("Learning Rate = 0.01")
plt.savefig("SLR001")
```

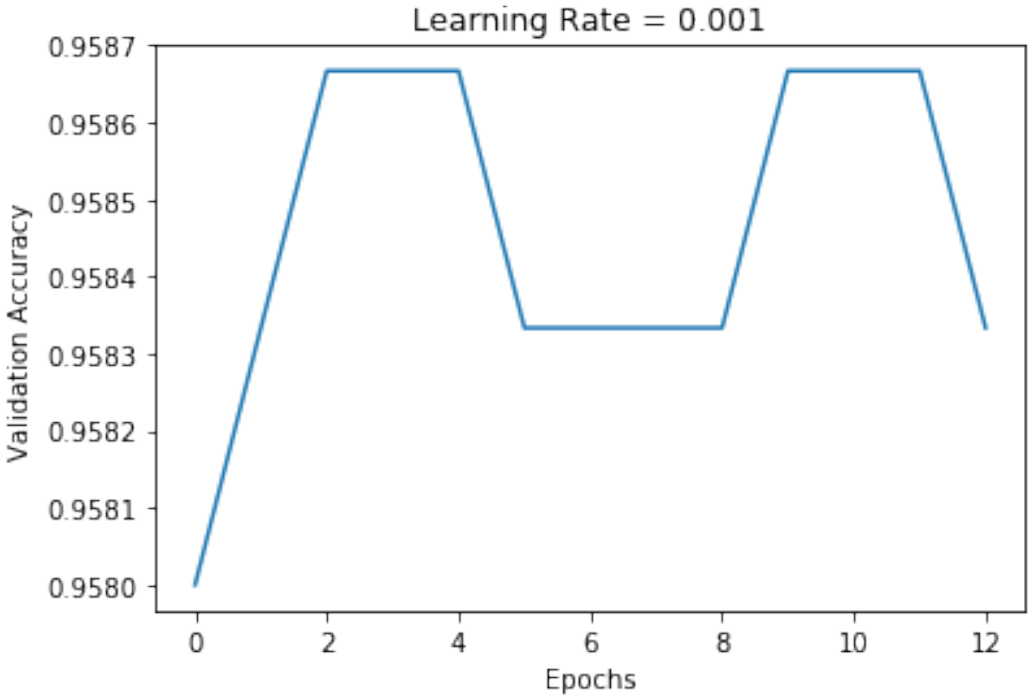


Accuracy stopped increasing at epoch 3. Training 13 epochs for learning_rate = 0.001

[9]:
```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

H3 = model.fit(
    x=train_image1D,
    y=train_label1D,
    shuffle=True,
    batch_size=BS,
    epochs=13,
    validation_data=(validate_image1D, validate_label1D),
    verbose=0
)
```

[10]:
```
plt.plot(H3.history["val_sparse_categorical_accuracy"])
plt.xlabel('Epochs')
plt.ylabel("Validation Accuracy")
plt.title("Learning Rate = 0.001")
plt.savefig("SLR0001")
```



[11]:
```
model.evaluate(x=test_images,y=test_labels,batch_size=BS)
```

```
100/100 [==============================] - 0s 549us/step - loss: 0.1942 -
sparse_categorical_accuracy: 0.9520
```

[11]:
```
[0.19416680932044983, 0.9520000219345093]
```