assignment_knn

January 28, 2021

1 ECE 176 Assignment 1: KNN

For this part of assignment, you are tasked to implement KNN algorithm and test it on the a subset of CIFAR10 dataset.

You sould run the whole notebook and answer the question in the notebook.

[1]:
```python
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt
```

1.1 Prepare Dataset

Since CIFAR10 is a relative large dataset, and KNN is quite time-consuming method, we only a small sub-set of CIFAR10 for KNN part

[2]:
```python
from data_processing import get_cifar10_data

# Use a subset of CIFAR10 for KNN assignments
dataset = get_cifar10_data(
    subset_train = 5000,
    subset_val = 250,
    subset_test = 500
)

print(dataset.keys())
print("Training Set Data  Shape: ", dataset['x_train'].shape)
print("Training Set Label Shape: ", dataset['y_train'].shape)
```

```
dict_keys(['x_train', 'y_train', 'x_val', 'y_val', 'x_test', 'y_test'])
Training Set Data  Shape:  (5000, 3072)
Training Set Label Shape:  (5000,)
```

1.2 Implementation (60%)

You need to implement the KNN method in `algorithms/knn.py`. You need to fill in the prediction function(since the training of KNN is just remembering the training set).

For KNN implementation, you are tasked to implement two version of it.

- Two Loop Version: use one loop to iterate through training samples and one loop to iterate through test samples
- One Loop Version: use one loop to iterate through test samples and use broadcast feature of numpy to calculate all the distance at once

Note: It is possible to build a Fully Vectorized Version without explicit for loop to calculate the distance, but you do not have to do it in this assignment.

For distance function, in this assignment, we use Euclidean distance between samples.

[3]:
```python
from algorithms import KNN
knn = KNN(
    num_class=10
)
knn.train(
    x_train=dataset['x_train'],
    y_train=dataset['y_train'],
    k=5
)
```

1.2.1 Compare the time consumption of different method

In this section, you will test your different implementation of KNN method, and compare their speed.

[4]:
```python
from evaluation import get_classification_accuracy
```

Two Loop Version:

[5]:
```python
import time
c_t = time.time()
prediction = knn.predict(
    dataset["x_test"],
    loop_count=2
)
print("Two Loop Prediction Time:", time.time()-c_t)

test_acc = get_classification_accuracy(prediction, dataset["y_test"])
print("Test Accuracy:", test_acc)
```

```
Two Loop Prediction Time: 47.15773677825928
Test Accuracy: 0.278
```

One Loop Version

[6]:
```python
import time
c_t = time.time()
prediction = knn.predict(
    dataset["x_test"], k=1,
    loop_count=1
)
print("One Loop Prediction Time:", time.time()-c_t)

test_acc = get_classification_accuracy(prediction, dataset["y_test"])
print("Test Accuracy:", test_acc)
```

```
One Loop Prediction Time: 66.04263591766357
Test Accuracy: 0.274
```

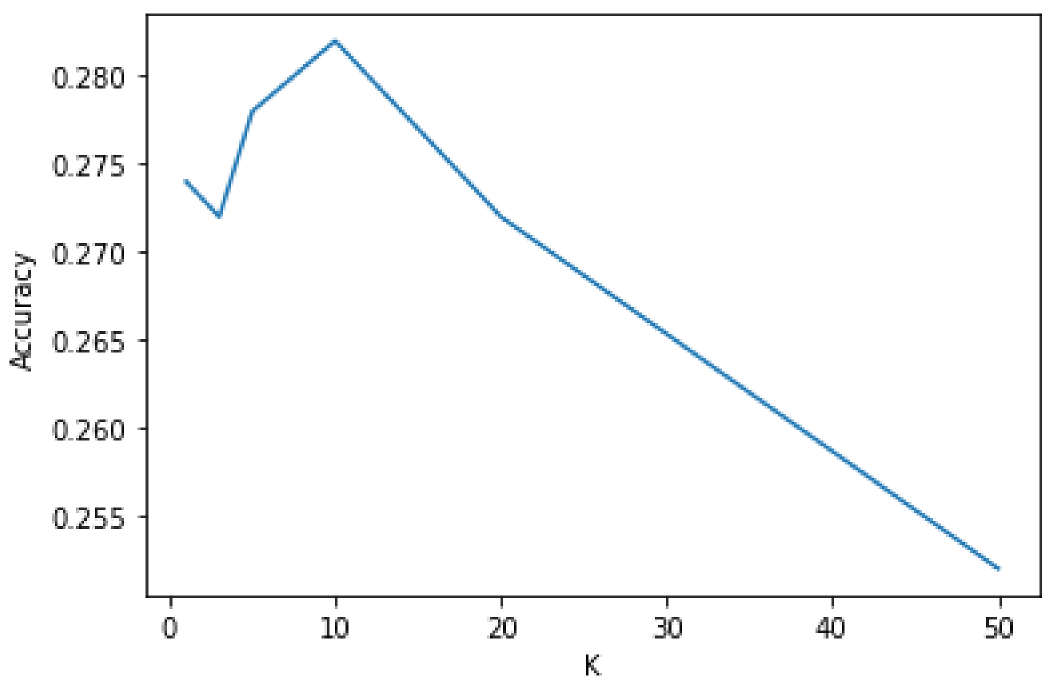
Your different implementation should output the exact same result

1.3 Test different Hyper-parameter(20%)

For KNN, there is only one hyper-parameter of the algorithm: How many nearest neighbour to use(**K**).

Here, you are provided the code to test different k for the same dataset.

[7]:
```
accuracies = []

k_candidates = [1,3,5,10,20,50]
for k_cand in k_candidates:
    prediction = knn.predict(
        x_test=dataset["x_test"],
        k=k_cand,
        loop_count=2
    )
    acc = get_classification_accuracy(prediction, dataset["y_test"])
    accuracies.append(acc)
plt.ylabel("Accuracy")
plt.xlabel("K")
plt.plot(k_candidates, accuracies)
plt.show()
```



1.3.1 Inline Question 1:

Please describe the output result you get, and provide some explanation as well.

1.3.2 Your Answer:

Put Your Answer Here

[8]:
```
#As one can see with the graph above, it seems that the best k value for this␣
↪dataset is k=10 as the accuracy maxes out
#at around 28% and starts to go down after. It makes sense that the accuracy␣
↪would go up as k goes up initially as there
#are more datapoints that the formula can compare each test datapoint to come␣
↪up with a better prediction. However, as k
#gets larger than 10 to 20 and 50, the accuracy goes down. This is because we␣
↪can infer that the training dataset is
#clustered locally in about 10 datapoints for each class. As for k=20 the␣
↪accuracy drops therefore at k=20 and beyond it
#is evident that the knn algorithm starts looking at training labels that are␣
↪more densely the incorrect label.
```

1.4 Try different feature representation(20%)

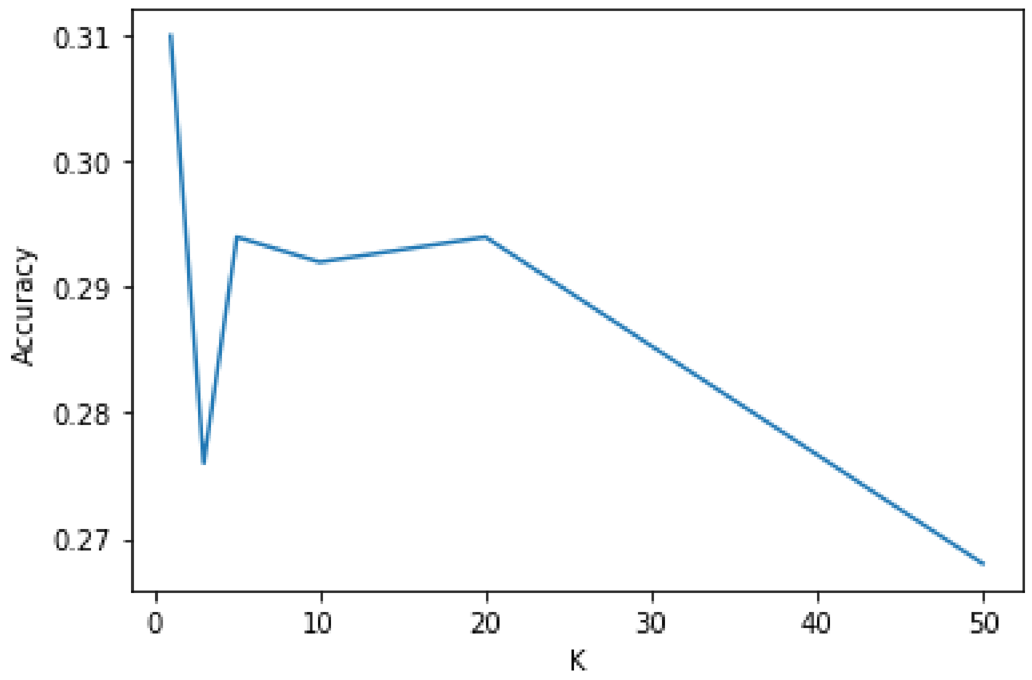
Since machine learning method rely heavily on the feature extraction, you will see how different feature representation affect the performance of the algorithm in this section.

You are provided the code about using **HOG** descriptor to represent samples in the notebook.

[9]:
```
from data_processing import get_cifar10_data
from data_processing import HOG_preprocess
from functools import partial

# Use a subset of CIFAR10 for KNN assignments
hog_p_func = partial(
    HOG_preprocess, orientations=9, pixels_per_cell=(4, 4),
    cells_per_block=(1, 1), visualize=False, multichannel=True
)
dataset = get_cifar10_data(
    feature_process=hog_p_func,
    subset_train = 5000,
    subset_val = 250,
    subset_test = 500
)
```

```
Start Processing
Processing Time: 12.509405612945557
```

[10]:
```
knn = KNN(
    num_class=10
)
knn.train(
    x_train=dataset['x_train'],
    y_train=dataset['y_train'],
    k=5,
)
accuracies = []

k_candidates = [1,3,5,10,20,50]
for k_cand in k_candidates:
    prediction = knn.predict(x_test=dataset["x_test"], k=k_cand)
    acc = get_classification_accuracy(prediction, dataset["y_test"])
    accuracies.append(acc)

plt.ylabel("Accuracy")
plt.xlabel("K")
plt.plot(k_candidates, accuracies)
plt.show()
```



1.4.1 Inline Question 2:

Please describe the output result you get, compare with the result you get in the previous section, and provide some explanation as well.

1.4.2 Your Answer:

Put Your Answer Here

[11]:
```
#It seems that the knn algorithm after using HOG works best at k=1 and drops at␣
↪k=3 and goes back up at k=5 and drop at k=50.
#This is different from the previous section results as k=10 had the best␣
↪accuracy and the accuracy dropped after that. This
#is probably because the HOG extracts the orientations of the gradients of all␣
↪the images and since our dataset are objects
#with discerning shapes and edges that have uniquely oriented gradients, the␣
↪HOG extraction would help our KNN determine
#the testing labels much better than without HOG extraction
```

assignment_linear

January 28, 2021

1 ECE 176 Assignment 1: Linear Regression

For this part of assignment, you are tasked to implement a linear regression algorithm for multiclass classification and test it on the CIFAR10 dataset.

You sould run the whole notebook and answer the questions in the notebook.

CIFAR 10 dataset contains 32x32x3 RGB images of 10 distinct cateogaries, and our aim is to predict which class the image belongs to

TO SUBMIT: PDF of this notebook with all the required outputs and answers.

[2]:
```
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from data_processing import get_cifar10_data

# Use a subset of CIFAR10 for the assignment
dataset = get_cifar10_data(
    subset_train = 5000,
    subset_val = 250,
    subset_test = 500,
)

print(dataset.keys())
print("Training Set Data  Shape: ", dataset['x_train'].shape)
print("Training Set Label Shape: ", dataset['y_train'].shape)
print("Validation Set Data  Shape: ", dataset['x_val'].shape)
print("Validation Set Label Shape: ", dataset['y_val'].shape)
print("Test Set Data  Shape: ", dataset['x_test'].shape)
print("Test Set Label Shape: ", dataset['y_test'].shape)
```

```
dict_keys(['x_train', 'y_train', 'x_val', 'y_val', 'x_test', 'y_test'])
Training Set Data  Shape:  (5000, 3072)
Training Set Label Shape:  (5000,)
Validation Set Data  Shape:  (250, 3072)
Validation Set Label Shape:  (250,)
Test Set Data  Shape:  (500, 3072)
Test Set Label Shape:  (500,)
```

[3]:
```
x_train = dataset['x_train']
y_train = dataset['y_train']
x_val = dataset['x_val']
y_val = dataset['y_val']
x_test = dataset['x_test']
y_test = dataset['y_test']
```

[3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
samples_per_class = 7

def visualize_data(dataset, classes, samples_per_class):
    num_classes = len(classes)
    for y, cls in enumerate(classes):
      idxs = np.flatnonzero(y_train == y)
      idxs = np.random.choice(idxs, samples_per_class, replace=False)
      for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(dataset[idx])
        plt.axis('off')
        if i == 0:
          plt.title(cls)
    plt.show()

visualize_data(x_train.reshape(5000, 3, 32, 32).transpose(0, 2, 3, 1), classes,␣
 ↪samples_per_class)
```

2 Linear Regression for multi-class classification

A Linear Regression Algorithm has 2 hyperparameters that you can experiment with: - **Learning rate** - controls how much we change the current weights of the classifier during each update. We set it at a default value of 0.5, and later you are asked to experiment with different values. We recommend looking at the graphs and observing how the performance of the classifier changes with different learning rate. - **Number of Epochs** - An epoch is a complete iterative pass over all of the data in the dataset. During an epoch we predict a label using the classifier and then update the weights of the classifier according the linear classifier update rule for each sample in the training set. We evaluate our models after every 10 epochs and save the accuracies, which are later used to plot the training, validation and test VS epoch curves. - **Weight Decay** - Regularization can be used to constrain the weights of the classifier and prevent their values from blowing up. Regularization helps in combatting overfitting. You will be using the 'weight_decay' term to introduce regularization in the classifier.

2.0.1 Implementation (50%)

You first need to implement the Linear Regression method in `algorithms/linear_regression.py`. You need to fill in the training function as well as the prediction function.

[4]:
```
# Import the algorithm implementation (TODO: Complete the Linear Regression in␣
 ↪algorithms/linear_regression.py)
from algorithms import Linear
from evaluation import get_classification_accuracy
num_classes = 10 #Cifar10 dataset has 10 different classes
```

```
# Initialize hyper-parameters
learning_rate = 0.001 # You will be later asked to experiment with different␣
↪learning rates and report results
num_epochs_total = 1000 # Total number of epochs to train the classifier
#for num_epochs_total I changed it to 800 as thats when the test accuracy was␣
↪highest for the visualization part at the end
epochs_per_evaluation = 10 # Epochs per step of evaluation; We will evaluate␣
↪our model regularly during training
N, D = dataset['x_train'].shape # Get training data shape, N: Number of␣
↪examples, D:Dimensionality of the data
weight_decay = 0.0

# Insert additional scalar term 1 in the samples to account for the bias as␣
↪discussed in class
x_train = np.insert(x_train, D, values = 1, axis = 1)
x_val = np.insert(x_val, D, values = 1, axis = 1)
x_test = np.insert(x_test, D, values = 1, axis = 1)
```

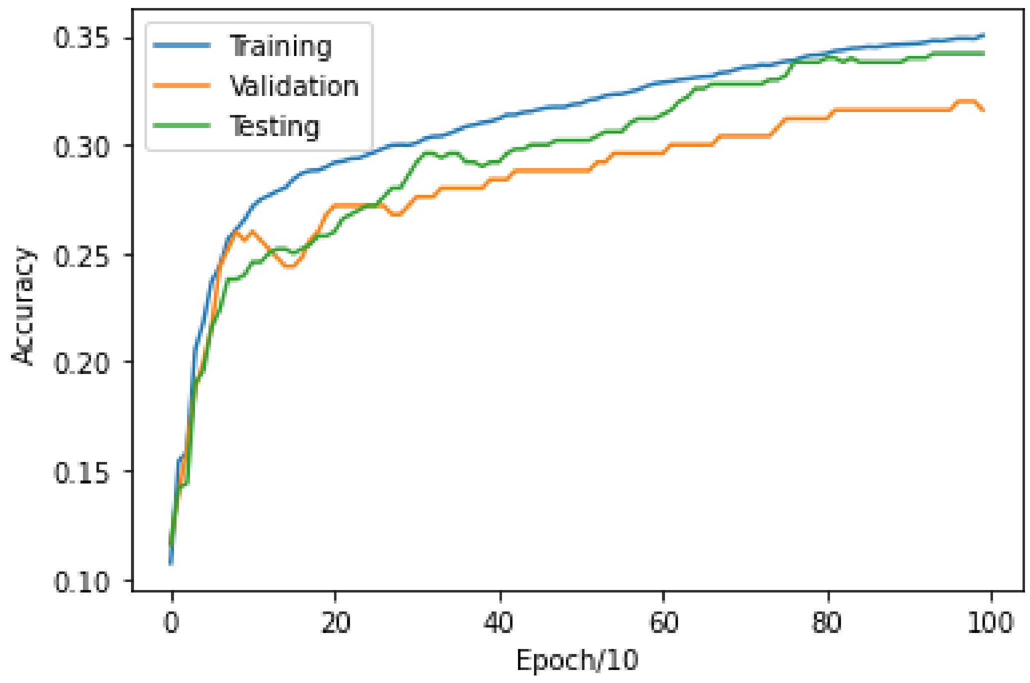
[5]:
```
# Training and evaluation function -> Outputs accuracy data

def train(learning_rate_, weight_decay_):
    # Create a linear regression object
    linear_regression = Linear(num_classes, learning_rate_,␣
↪epochs_per_evaluation, weight_decay_)

    # Randomly initialize the weights and biases
    weights = np.random.randn(num_classes, D+1) * 0.0001

    train_accuracies, val_accuracies, test_accuracies = [], [], []

    #Train the classifier
    for _ in range(int(num_epochs_total/epochs_per_evaluation)):
        # Train the classifier on the training data
        weights = linear_regression.train(x_train, y_train, weights)

        # Evaluate the trained classifier on the training dataset
        y_pred_train = linear_regression.predict(x_train)
        train_accuracies.append(get_classification_accuracy(y_pred_train,␣
↪y_train))

        # Evaluate the trained classifier on the validation dataset
        y_pred_val = linear_regression.predict(x_val)
        val_accuracies.append(get_classification_accuracy(y_pred_val, y_val))

        # Evaluate the trained classifier on the test dataset
        y_pred_test = linear_regression.predict(x_test)
        test_accuracies.append(get_classification_accuracy(y_pred_test, y_test))

    return train_accuracies, val_accuracies, test_accuracies, weights
```

2.0.2 Plot the Accuracies vs epoch graphs

[6]:
```
import matplotlib.pyplot as plt

def plot_accuracies(train_acc, val_acc, test_acc):
    # Plot Accuracies vs Epochs graph for all the three
    epochs = np.arange(0, int(num_epochs_total/epochs_per_evaluation))
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch/10")
    plt.plot(epochs, train_acc, epochs, val_acc, epochs, test_acc)
    plt.legend(['Training', 'Validation', 'Testing'])
    plt.show()
```

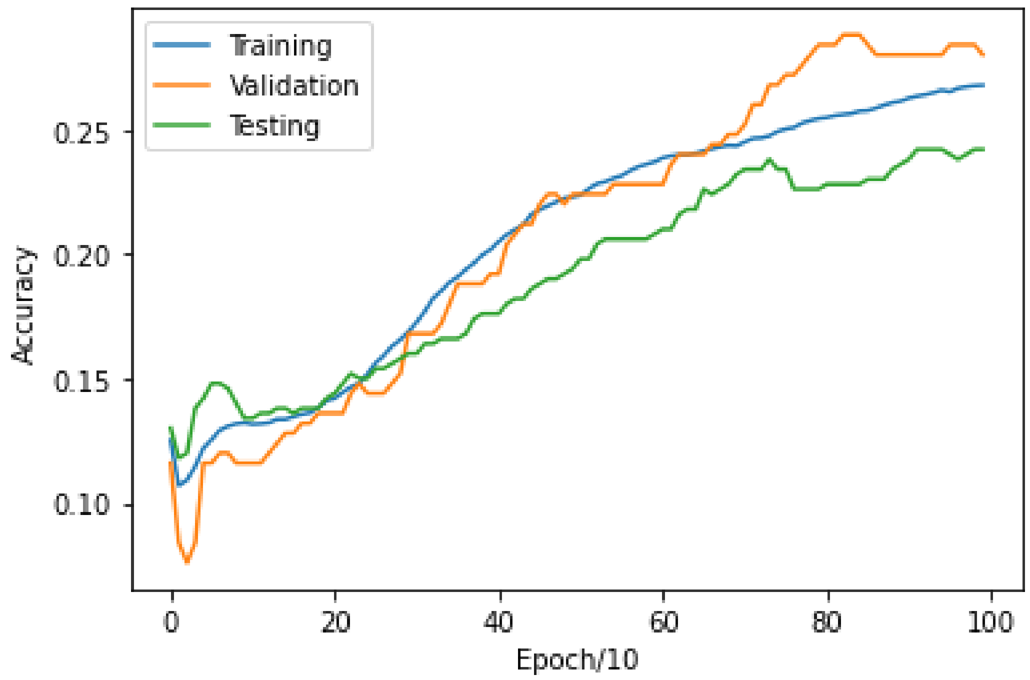
[11]:
```
# Run training and plotting for default parameter values as mentioned above
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
```

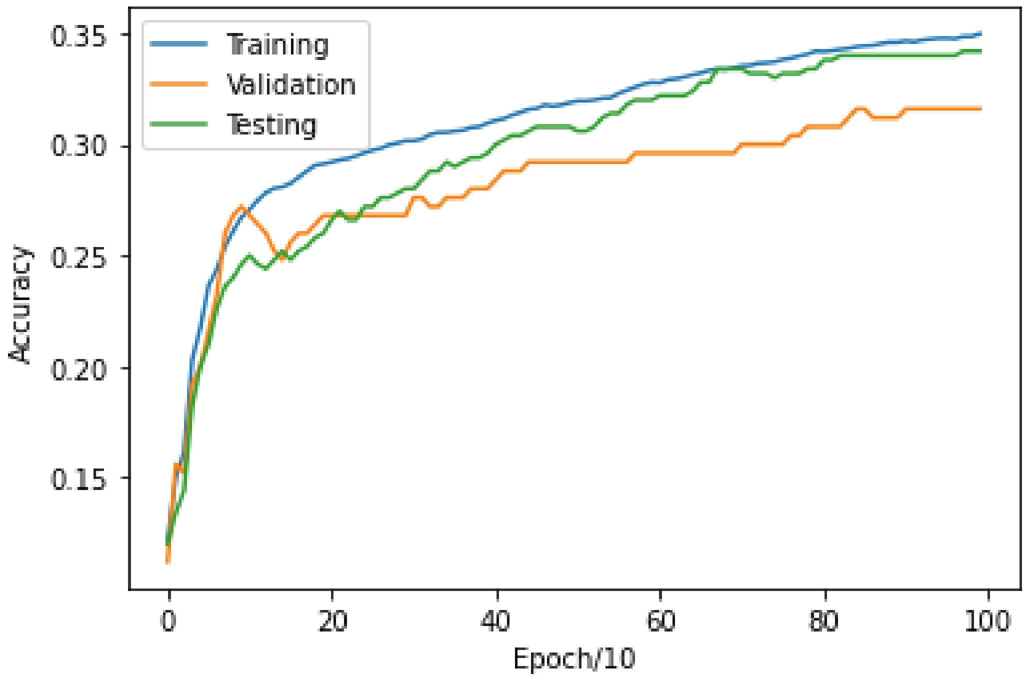
[12]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```

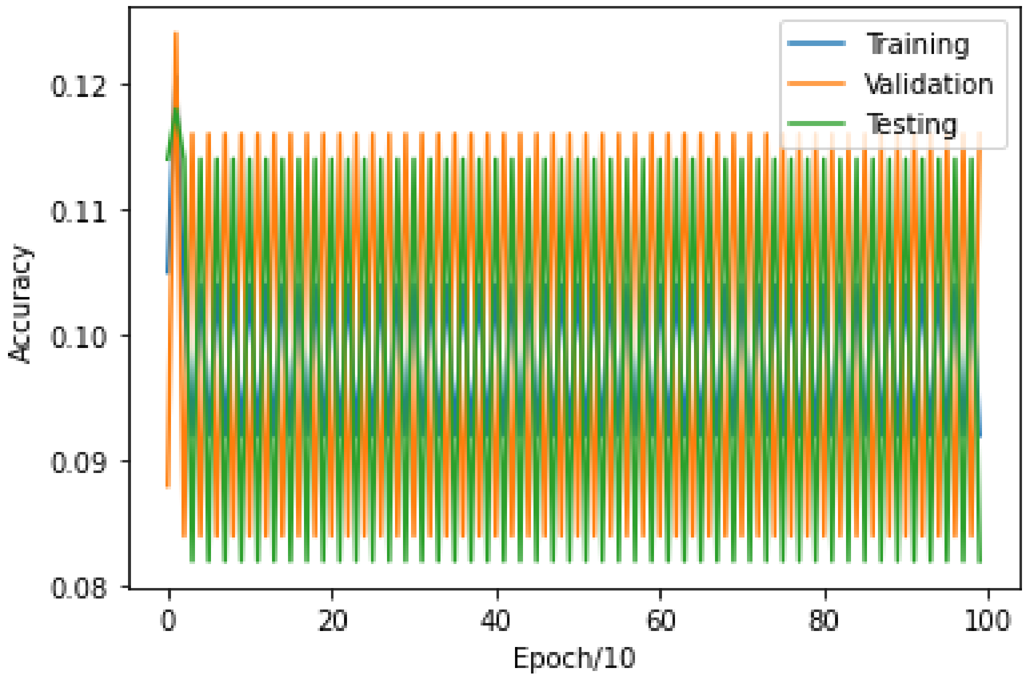


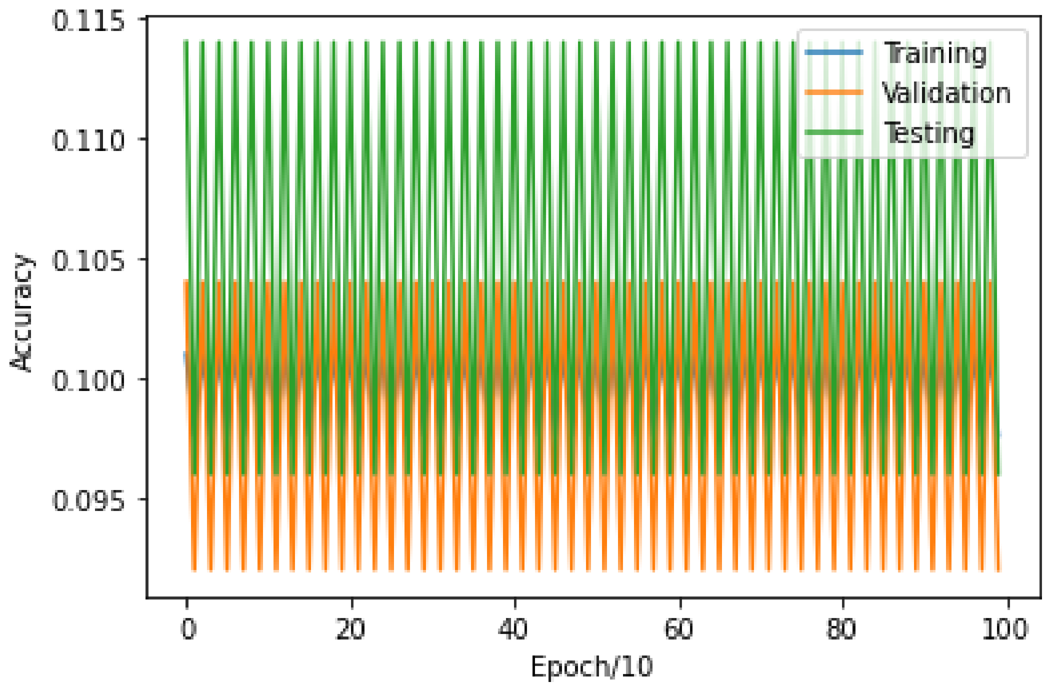
2.0.3 Try different learning rates and plot graphs for all (20%)

[8]:
```
# TODO
# Repeat the above training and evaluation steps for the following learning␣
↪rates and plot graphs
# You need to submit all 4 graphs along with this notebook pdf
learning_rates = [0.0001, 0.001, 0.01, 0.1]
weight_decay = 0.0 # No regularization for now
for i in range(4):
    t_ac, v_ac, te_ac, weights = train(learning_rates[i], weight_decay)
    plot_accuracies(t_ac, v_ac, te_ac)

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
↪ACHIEVE A BETTER PERFORMANCE

# for lr in learning_rates: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)
```




Training
Validation
Testing
Accuracy
Epoch/10
0.35
0.30
0.25
0.20
0.15
0
20
40
60
80
100


Training
Validation
Testing
Accuracy
Epoch/10
0.12
0.11
0.10
0.09
0.08
0
20
40
60
80
100



Inline Question 1. Which one of these learning rates (best_lr) would you pick to train your model? Please Explain why.

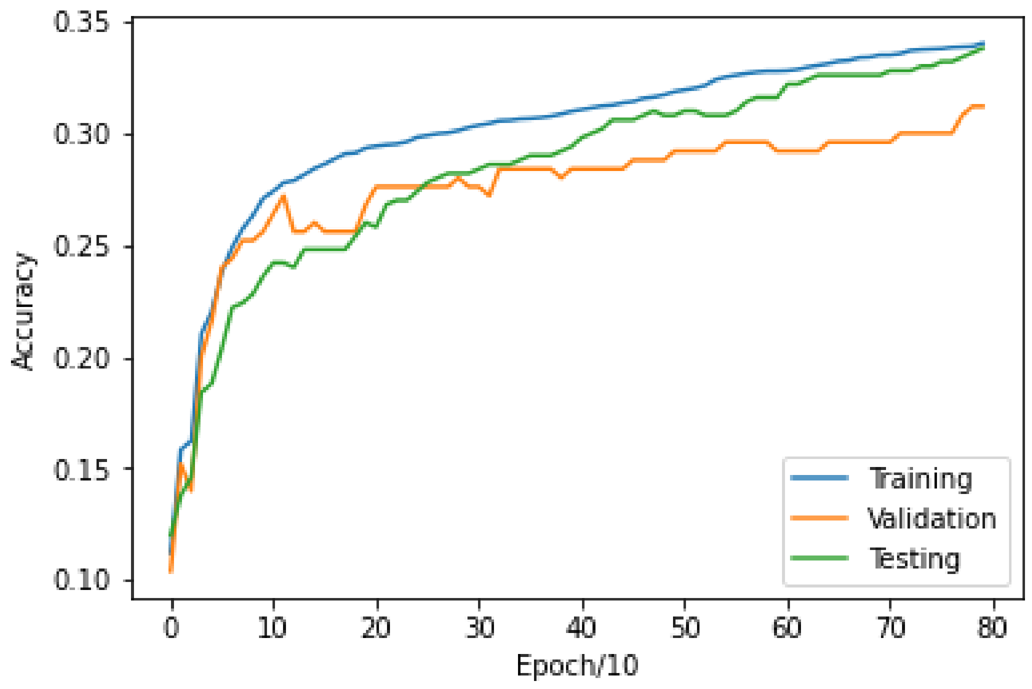
Your Answer:

```
[25]: #The one with 0.001 has the best learning rate for this model in my opinion as␣
       ↪it has the best test accuracy. The 0.0001
      #learning rate is a too low learning rate and converges too slowly while the 0.
       ↪01 and 0.1 learning rates are too high as the graph
      #shows that the accuracy oscillates a lot and misses the global minimum for␣
       ↪gradient descent.
```

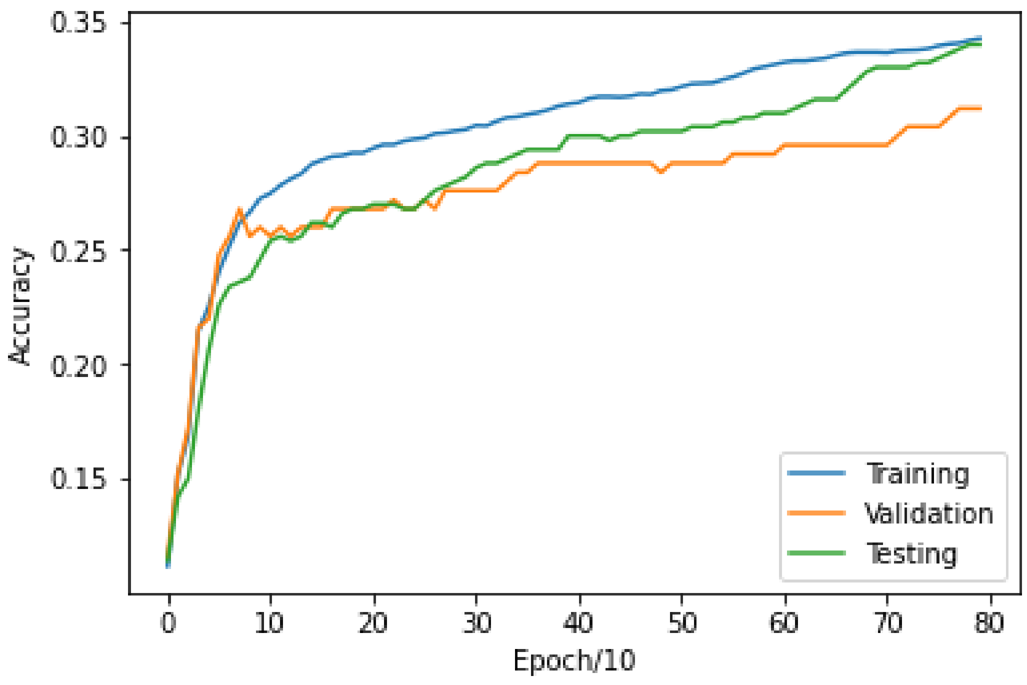
2.0.4 Regularization: Try different weight decay and plot graphs for all (20%)

```
[22]: # Initialize a non-zero weight_decay (Regulzarization constant) term and repeat␣
       ↪the training and evaluation
      # Use the best learning rate as obtained from the above excercise, best_lr
      best_lr=0.001
      weight_decays = [0.0, 0.1, 1, 10, 100]
      for i in range(5):
          t_ac, v_ac, te_ac, weights = train(best_lr, weight_decays[i])
          plot_accuracies(t_ac, v_ac, te_ac)
          print(t_ac[-1], v_ac[-1], te_ac[-1]) #print accuracies to compare graphs␣
       ↪better
```

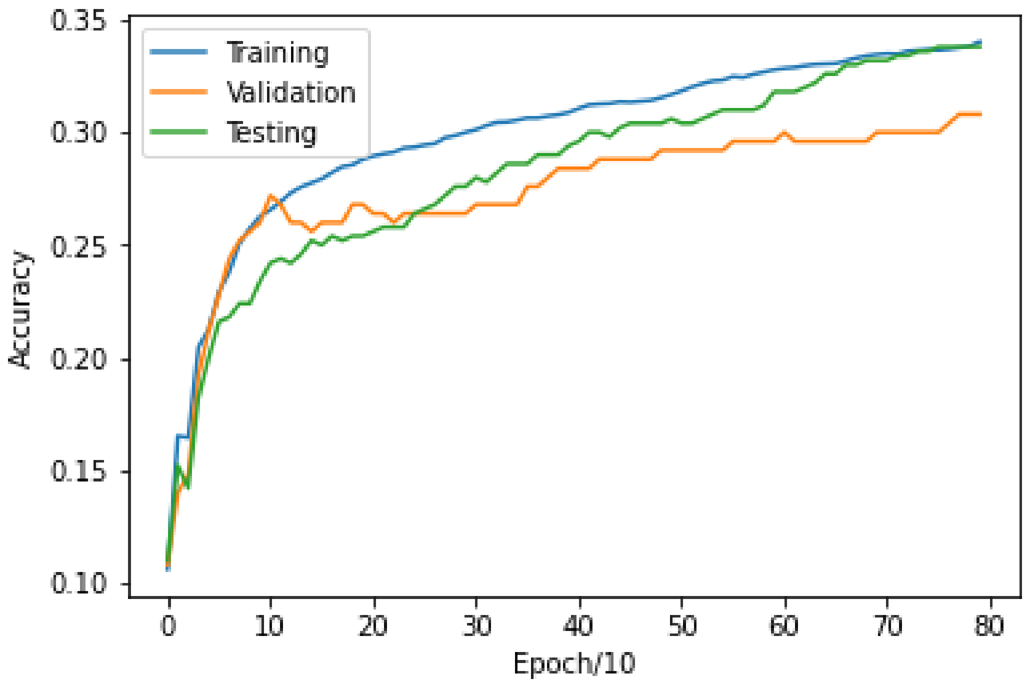
```
# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
↪ACHIEVE A BETTER PERFORMANCE

# for weight_decay in weight_decays: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(best_lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)
```



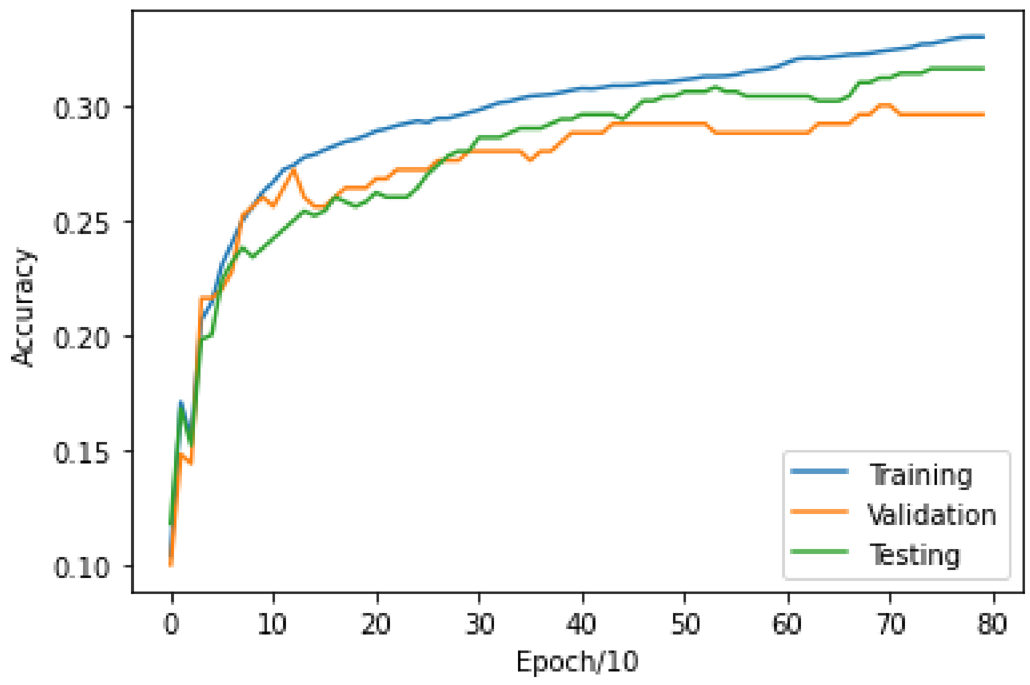
```
0.3402 0.312 0.338
```



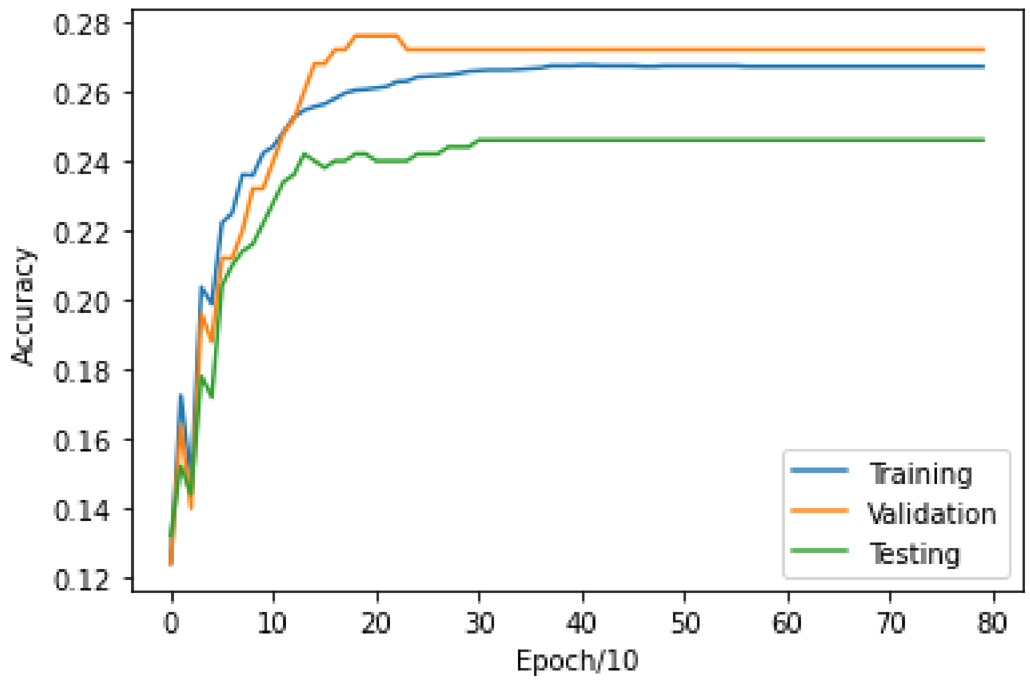
```
0.3426 0.312 0.34
```



```
0.3402 0.308 0.338
```



```
0.3298 0.296 0.316
```



```
0.2672 0.272 0.246
```

Inline Question 2. Discuss underfitting and overfitting as observed in the 5 graphs obtained by changing the regularization. Which weight_decay term gave you the best classifier performance? HINT: Do not just think in terms of best training set performance, keep in mind that the real utility of a machine learning model is when it performs well on data it has never seen before

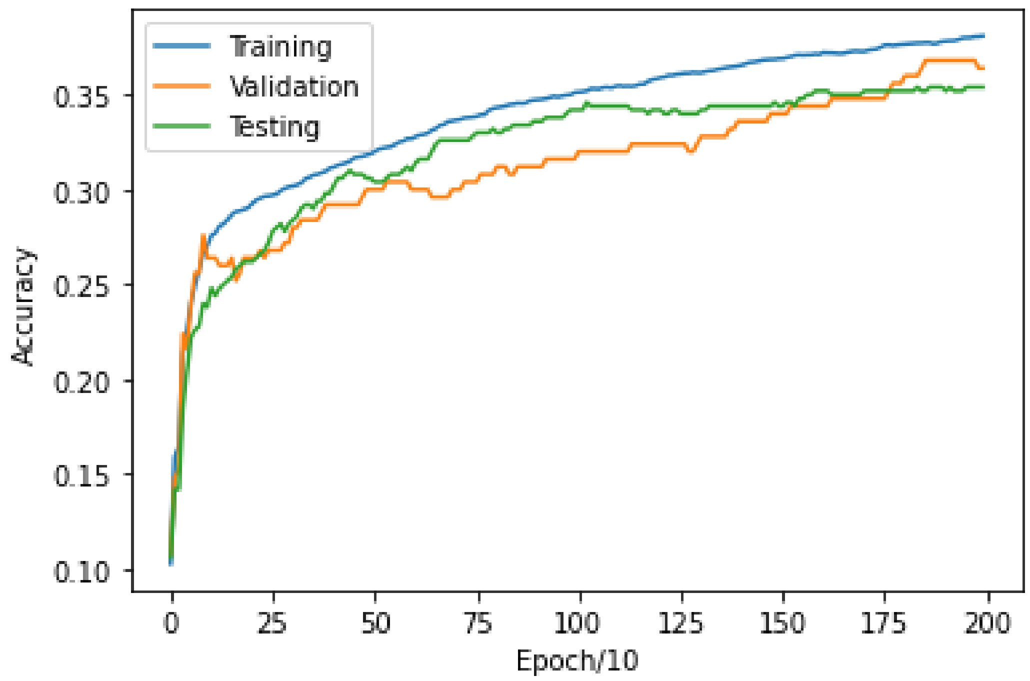
Your Answer:

[24]:
```
#Overfitting occurs in weight_decay=0 as the difference between test accuracy␣
↪and training accuracy is bigger than the
#difference when weight_decay=0.1. Underfitting occurs when weight_decay=100 as␣
↪the test accuracy is much lower than the
#training accuracy by around 2.1%. I would use a weight_decay of 0.1 as it has␣
↪the smallest difference between test accuracy
#and training accuracy and also performs the best on test accuracy which is new␣
↪data for the model.
```

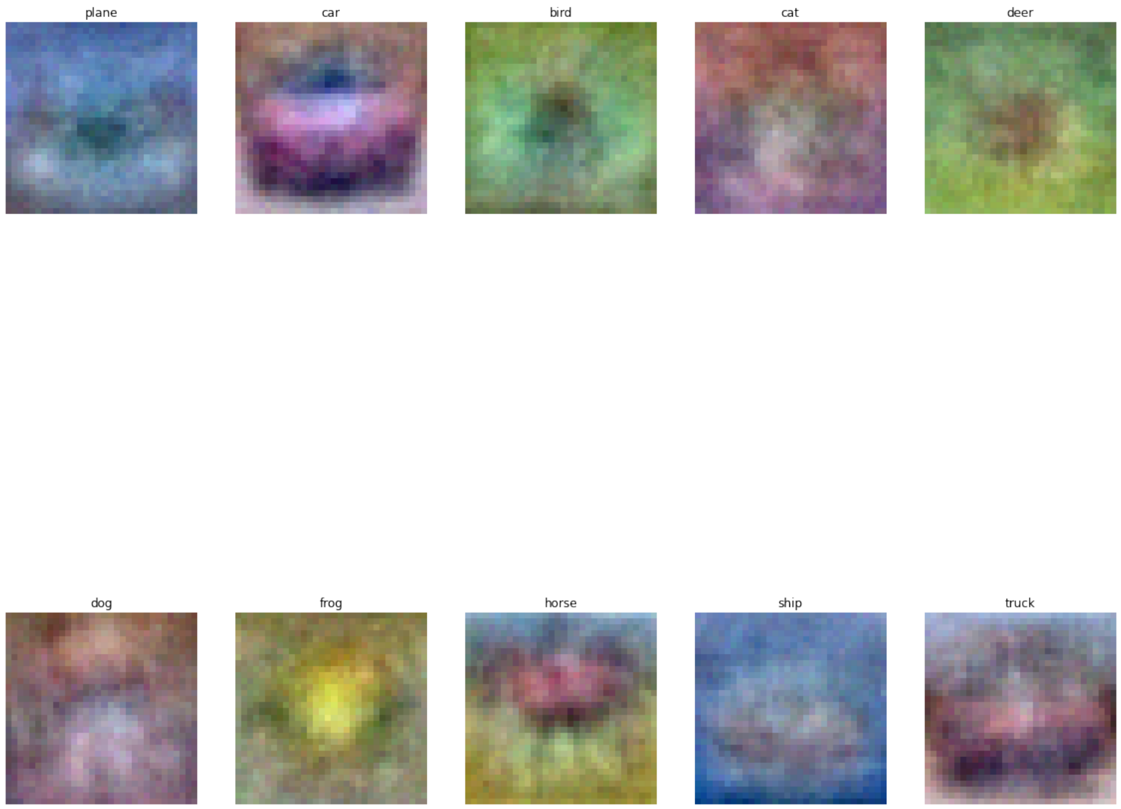
2.0.5 Visualize the filters (10%)

[9]:
```
# These visualizations will only somewhat make sense if your learning rate and␣
↪weight_decay parameters were
# properly chosen in the model. Do your best.

#below is my code to choose best 3 parameters by looking at the graphs of lr=0.
↪001 and weight_decay=0.1 and num_epochs_total=800
learning_rate = 0.001 # You will be later asked to experiment with different␣
↪learning rates and report results
num_epochs_total = 2000 # Total number of epochs to train the classifier
#for num_epochs_total I changed it to 800 as thats when the test accuracy was␣
↪highest for the visualization part at the end
epochs_per_evaluation = 10 # Epochs per step of evaluation; We will evaluate␣
↪our model regularly during training
N, D = dataset['x_train'].shape # Get training data shape, N: Number of␣
↪examples, D:Dimensionality of the data
weight_decay = 0.1
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
plot_accuracies(t_ac, v_ac, te_ac) # to see the accuracy with best results

w = weights[:, :-1]
print(weights.shape)
w = w.reshape(10, 3, 32, 32).transpose(0, 2, 3, 1)

w_min, w_max = np.min(w), np.max(w)

fig=plt.figure(figsize=(20, 20))
```

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
for i in range(10):
    fig.add_subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[i, :, :, :].squeeze() - w_min) / (w_max - w_min)
    #plt.imshow(wimg.astype('uint8'))
    plt.imshow(wimg.astype(int))
    plt.axis('off')
    plt.title(classes[i])
plt.show()

# TODO: Run this cell and Show filter visualizations for the best set of␣
↪weights you obtain. Report the 3 hyperparameters you used
# to obtain the best model.

# Be careful about choosing the 'weights' obtained from the correct trained␣
↪classifier
```



```
(10, 3073)
```



[20]:
```
#I chose epochs=2000, learning rate=0.001 and weight decay=1 because the␣
↪accuracy graph with those learning rate and weight
#decay has the best accuracy training and test. I chose 3000 for epochs because␣
↪the test accuracy graph converges at around
#2000 epochs when I tried testing out the model with 10000 epochs.
```

assignment_logistic

January 28, 2021

1 ECE 176 Assignment 1: Logistic Regression

For this part of assignment, you are tasked to implement a logistic regression algorithm for multi-class classification and test it on the CIFAR10 dataset.

You sould run the whole notebook and answer the questions in the notebook.

TO SUBMIT: PDF of this notebook with all the required outputs and answers.

[2]:
```python
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from data_processing import get_cifar10_data

# Use a subset of CIFAR10 for KNN assignments
dataset = get_cifar10_data(
    subset_train = 5000,
    subset_val = 250,
    subset_test = 500,
)

print(dataset.keys())
print("Training Set Data  Shape: ", dataset['x_train'].shape)
print("Training Set Label Shape: ", dataset['y_train'].shape)
print("Validation Set Data  Shape: ", dataset['x_val'].shape)
print("Validation Set Label Shape: ", dataset['y_val'].shape)
print("Test Set Data  Shape: ", dataset['x_test'].shape)
print("Test Set Label Shape: ", dataset['y_test'].shape)
```

```
dict_keys(['x_train', 'y_train', 'x_val', 'y_val', 'x_test', 'y_test'])
Training Set Data  Shape:  (5000, 3072)
Training Set Label Shape:  (5000,)
Validation Set Data  Shape:  (250, 3072)
Validation Set Label Shape:  (250,)
Test Set Data  Shape:  (500, 3072)
Test Set Label Shape:  (500,)
```

[3]:
```
x_train = dataset['x_train']
y_train = dataset['y_train']
x_val = dataset['x_val']
y_val = dataset['y_val']
x_test = dataset['x_test']
y_test = dataset['y_test']
```

2 Logistic Regression for multi-class classification

A Logistic Regression Algorithm has 3 hyperparameters that you can experiment with: - **Learning rate** - controls how much we change the current weights of the classifier during each update. We set it at a default value of 0.5, and later you are asked to experiment with different values. We recommend looking at the graphs and observing how the performance of the classifier changes with different learning rate. - **Number of Epochs** - An epoch is a complete iterative pass over all of the data in the dataset. During an epoch we predict a label using the classifier and then update the weights of the classifier according the linear classifier update rule for each sample in the training set. We evaluate our models after every 10 epochs and save the accuracies, which are later used to plot the training, validation and test VS epoch curves. - **Weight Decay** - Regularization can be used to constrain the weights of the classifier and prevent their values from blowing up. Regularization helps in combatting overfitting. You will be using the 'weight_decay' term to introduce regularization in the classifier.

The only way how a Logistic Regression based classification algorithm is different from a Linear Regression algorithm is that in the former we additionally pass the classifier outputs into a sigmoid function which squashes the output in the (0,1) range. Essentially these values then represent the probabilities of that sample belonging to class particular classes

2.0.1 Implementation (40%)

You need to implement the Linear Regression method in `algorithms/logistic_regression.py`. You need to fill in the sigmoid function, training function as well as the prediction function.

[4]:
```
# Import the algorithm implementation (TODO: Complete the Logistic Regression␣
 ↪in algorithms/logistic_regression.py)
from algorithms import Logistic
from evaluation import get_classification_accuracy
num_classes = 10 #Cifar10 dataset has 10 different classes

# Initialize hyper-parameters
learning_rate = 0.01 # You will be later asked to experiment with different␣
 ↪learning rates and report results
num_epochs_total = 1000 # Total number of epochs to train the classifier
epochs_per_evaluation = 10 # Epochs per step of evaluation; We will evaluate␣
 ↪our model regularly during training
N, D = dataset['x_train'].shape # Get training data shape, N: Number of␣
 ↪examples, D:Dimensionality of the data
weight_decay = 0.0
```

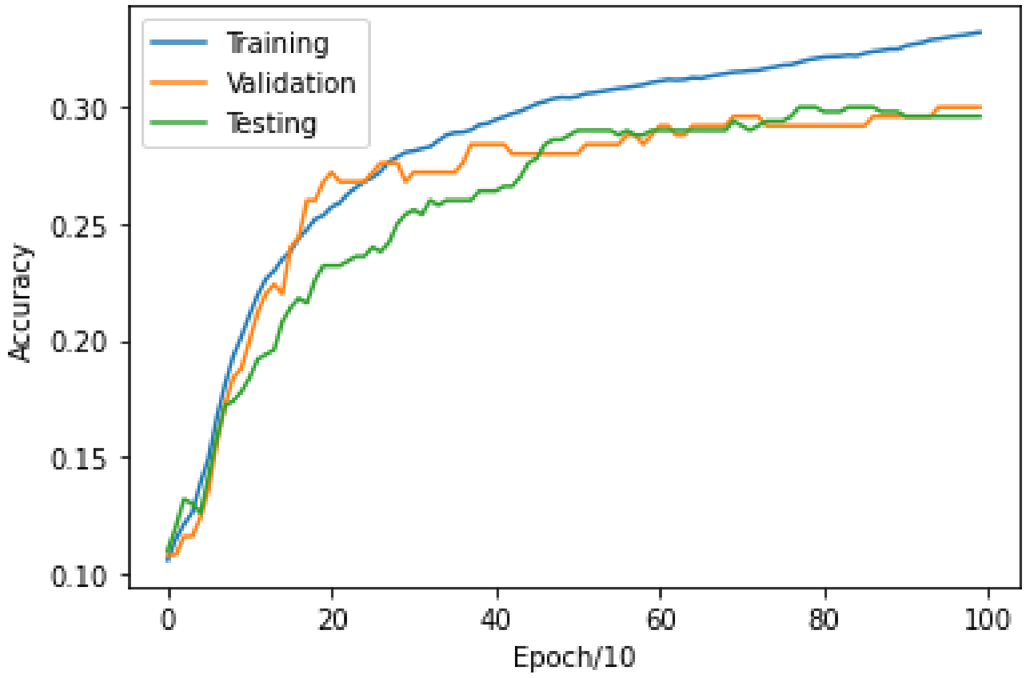
```
# Insert additional scalar term 1 in the samples to account for the bias as␣
↪discussed in class
x_train = np.insert(x_train, D, values = 1, axis = 1)
x_val = np.insert(x_val, D, values = 1, axis = 1)
x_test = np.insert(x_test, D, values = 1, axis = 1)
```

[5]:
```
# Training and evaluation function -> Outputs accuracy data

def train(learning_rate_, weight_decay_):
    # Create a linear regression object
    logistic_regression = Logistic(num_classes, learning_rate_,␣
↪epochs_per_evaluation, weight_decay_)

    # Randomly initialize the weights and biases
    weights = np.random.randn(num_classes, D+1) * 0.0001

    train_accuracies, val_accuracies, test_accuracies = [], [], []

    #Train the classifier
    for _ in range(int(num_epochs_total/epochs_per_evaluation)):
        # Train the classifier on the training data
        weights = logistic_regression.train(x_train, y_train, weights)

        # Evaluate the trained classifier on the training dataset
        y_pred_train = logistic_regression.predict(x_train)
        train_accuracies.append(get_classification_accuracy(y_pred_train,␣
↪y_train))

        # Evaluate the trained classifier on the validation dataset
        y_pred_val = logistic_regression.predict(x_val)
        val_accuracies.append(get_classification_accuracy(y_pred_val, y_val))

        # Evaluate the trained classifier on the test dataset
        y_pred_test = logistic_regression.predict(x_test)
        test_accuracies.append(get_classification_accuracy(y_pred_test, y_test))

    return train_accuracies, val_accuracies, test_accuracies, weights
```

[6]:
```
import matplotlib.pyplot as plt

def plot_accuracies(train_acc, val_acc, test_acc):
    # Plot Accuracies vs Epochs graph for all the three
    epochs = np.arange(0, int(num_epochs_total/epochs_per_evaluation))
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch/10")
    plt.plot(epochs, train_acc, epochs, val_acc, epochs, test_acc)
    plt.legend(['Training', 'Validation', 'Testing'])
    plt.show()
```

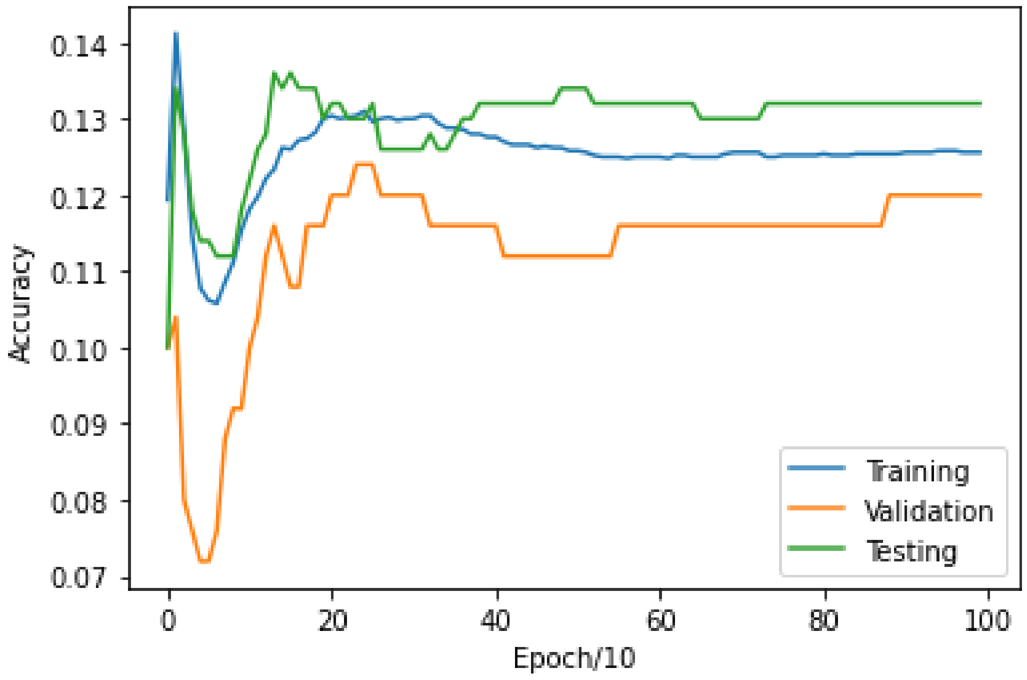
[8]:
```
# Run training and plotting for default parameter values as mentioned above
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
```

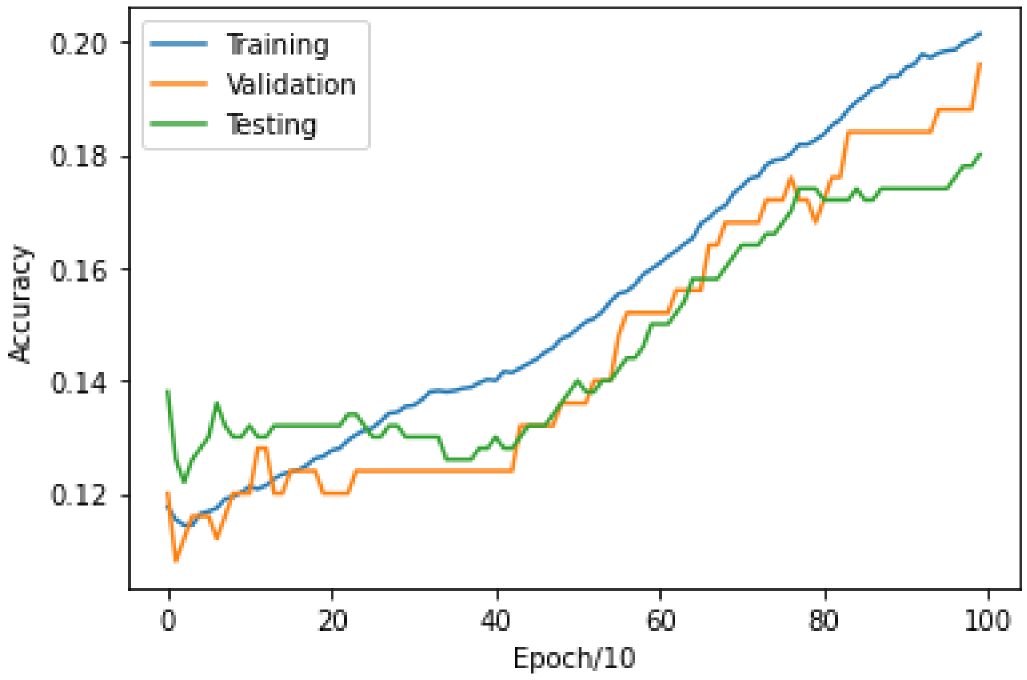
[9]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```

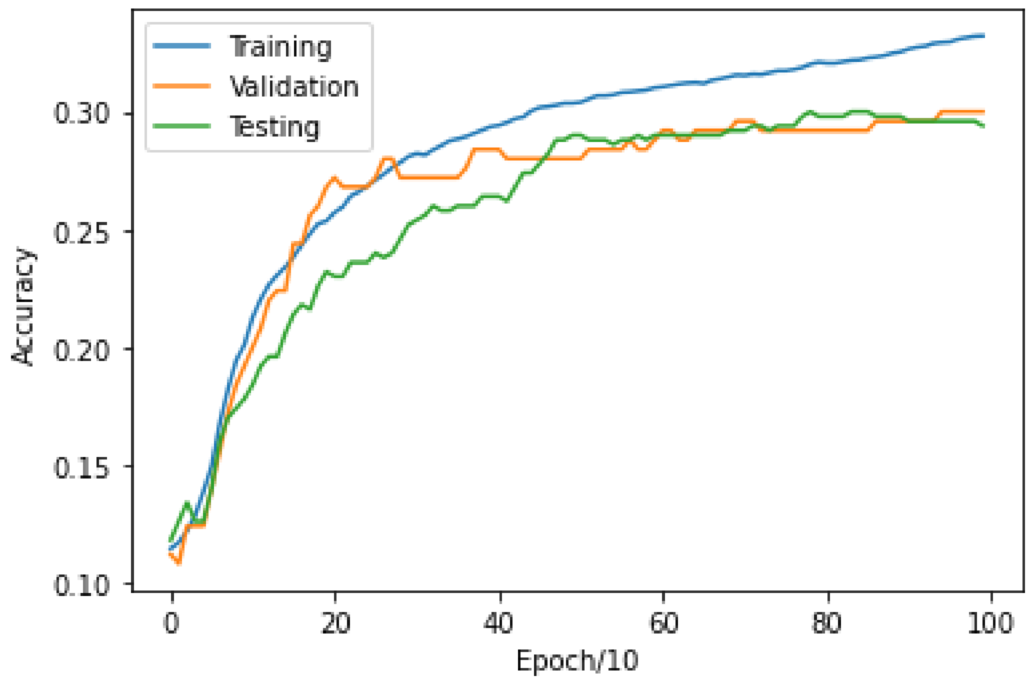


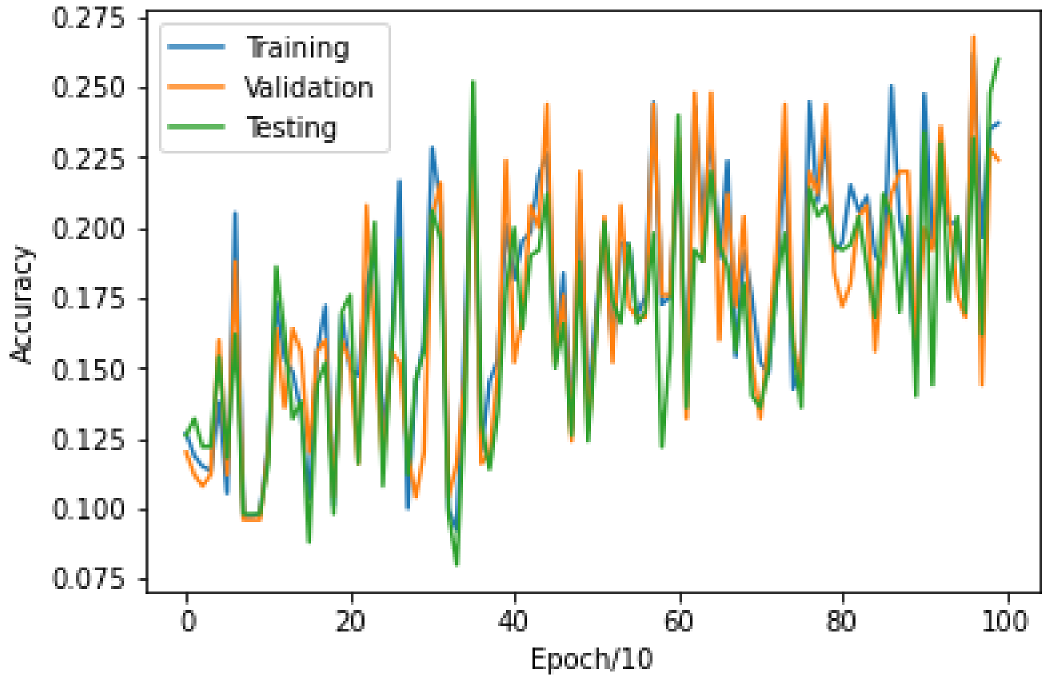
2.0.2 Try different learning rates and plot graphs for all (20%)

[8]:
```
# TODO
# Repeat the above training and evaluation steps for the following learning␣
 ↪rates and plot graphs
# You need to submit all 5 graphs along with this notebook pdf
learning_rates = [0.0001, 0.001, 0.01, 0.1]
weight_decay = 0.0 # No regularization for now
for i in range(4):
    t_ac, v_ac, te_ac, weights = train(learning_rates[i], weight_decay)
    plot_accuracies(t_ac, v_ac, te_ac)

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
 ↪ACHIEVE A BETTER PERFORMANCE

# for lr in learning_rates: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)
```









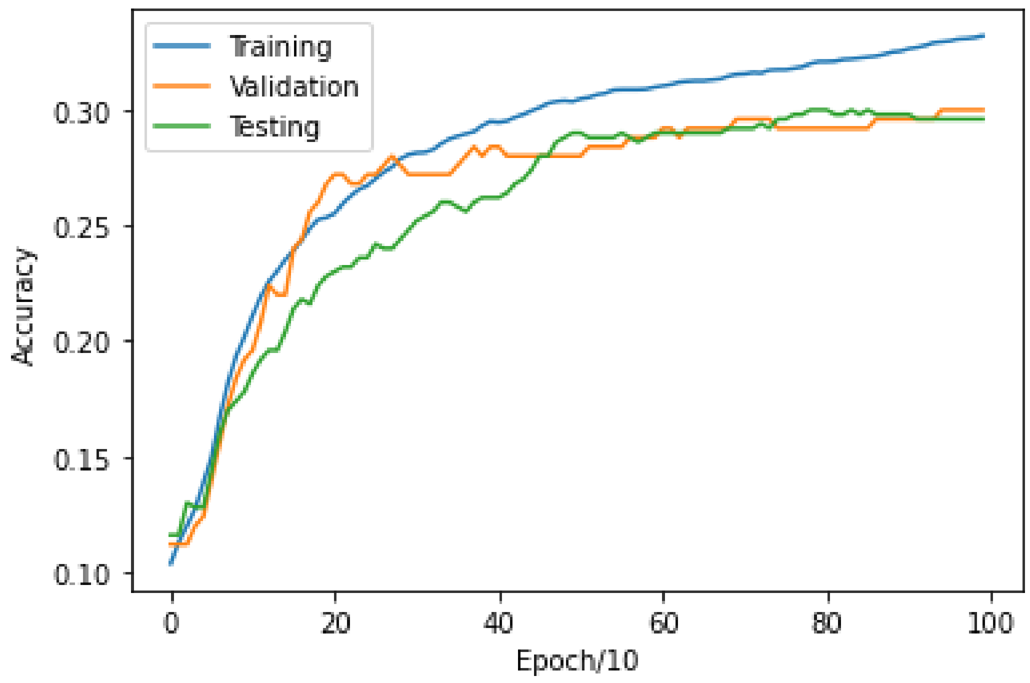
Inline Question 1. Which one of these learning rates (best_lr) would you pick to train your model? Please Explain why.

Your Answer:

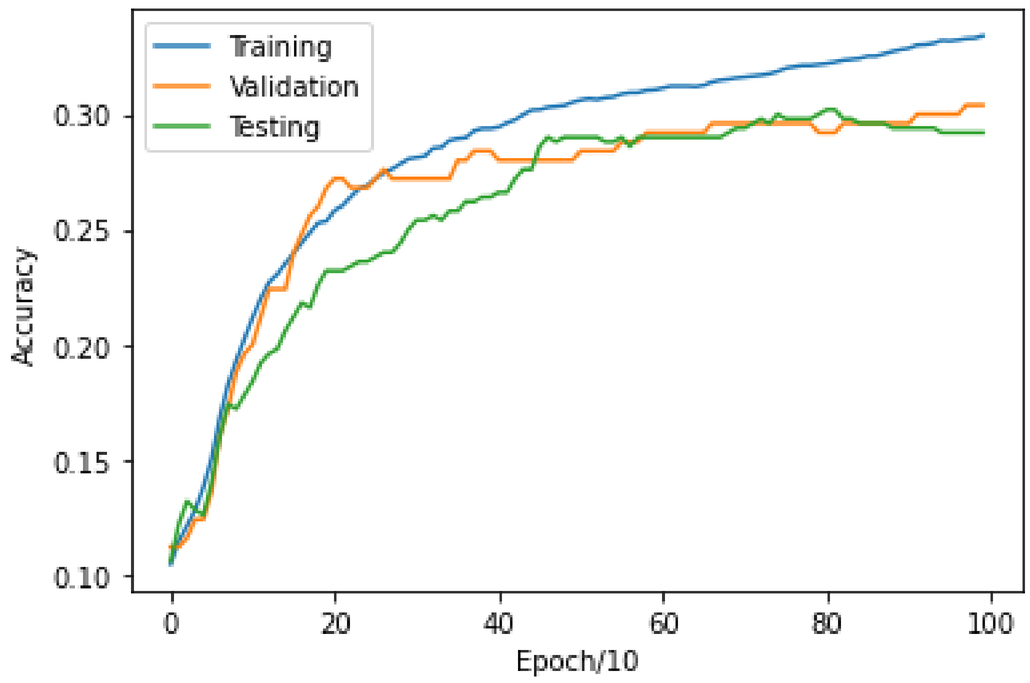
```
[13]: #I would pick the learning rate=0.01 as it has the highest test and training␣
      ↪accuracy out of all of them. The learning rates
      #smaller than that are too slow to converge to the best accuracy, and the ones␣
      ↪bigger than that are too large that they
      #overshoot the correct weights.
```

2.0.3 Regularization: Try different weight decay and plots graphs for all (20%)

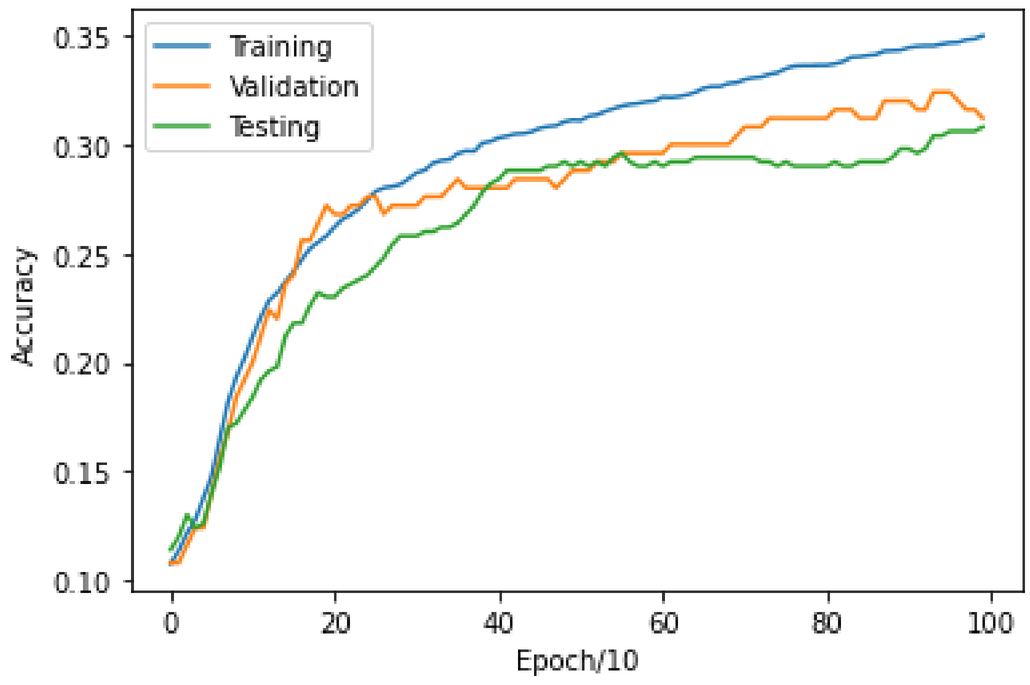
```
[7]: # Initialize a non-zero weight_decay (Regulzarization constant) term and repeat␣
     ↪the training and evaluation
     # Use the best learning rate as obtained from the above excercise, best_lr
     weight_decays = [0.0, 0.1, 1, 10, 100]
     best_lr=0.01
     for i in range(5):
         t_ac, v_ac, te_ac, weights = train(best_lr, weight_decays[i])
         plot_accuracies(t_ac, v_ac, te_ac)
         print(t_ac[-1], v_ac[-1], te_ac[-1]) #print accuracies to compare graphs␣
     ↪better



     # FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
     ↪ACHIEVE A BETTER PERFORMANCE

     # for weight_decay in weight_decays: Train the classifier and plot data
     # Step 1. train_accu, val_accu, test_accu = train(best_lr, weight_decay)
     # Step 2. plot_accuracies(train_accu, val_accu, test_accu)
```



```
0.332 0.3 0.296
```

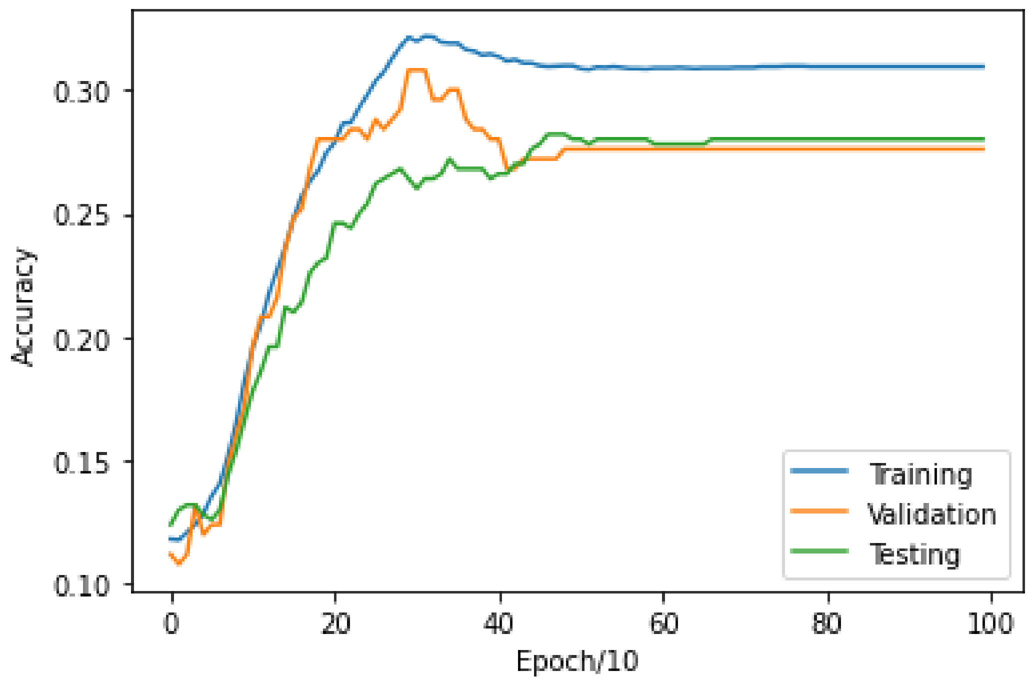


```
0.334 0.304 0.292
```



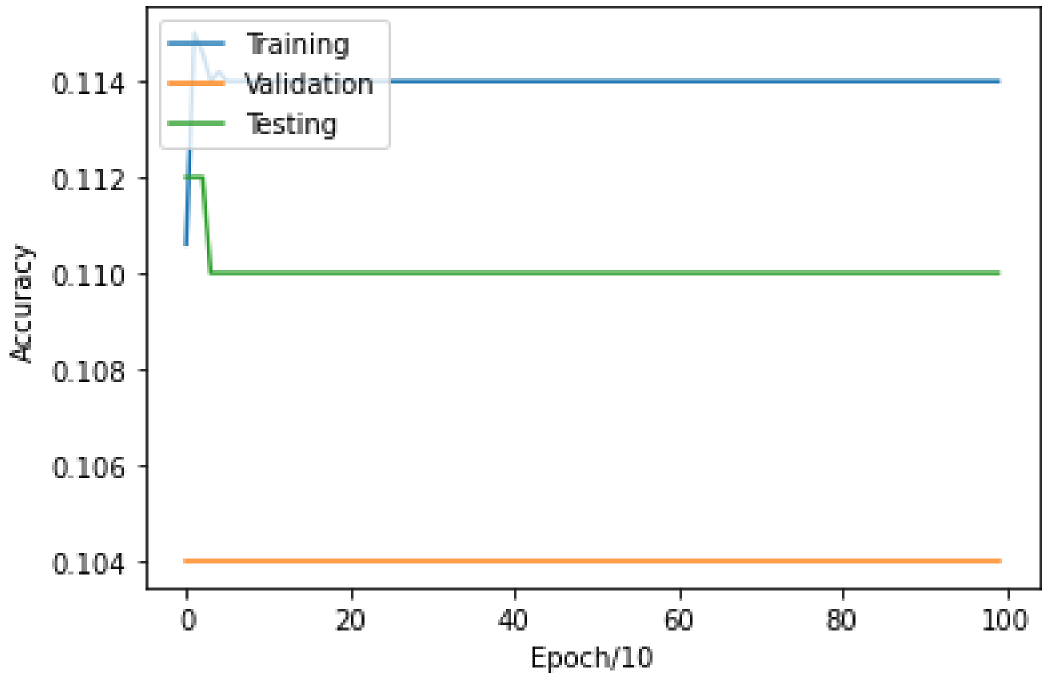
```
0.3498 0.312 0.308
```

```
/home/lsassone/algorithms/logistic_regression.py:33: RuntimeWarning: overflow
encountered in exp
  sigomode=1/(1+np.exp(-1*z))
```



```
0.3094 0.276 0.28
```

```
/home/lsassone/algorithms/logistic_regression.py:33: RuntimeWarning: overflow
encountered in exp
  sigomode=1/(1+np.exp(-1*z))
```



```
0.114 0.104 0.11
```

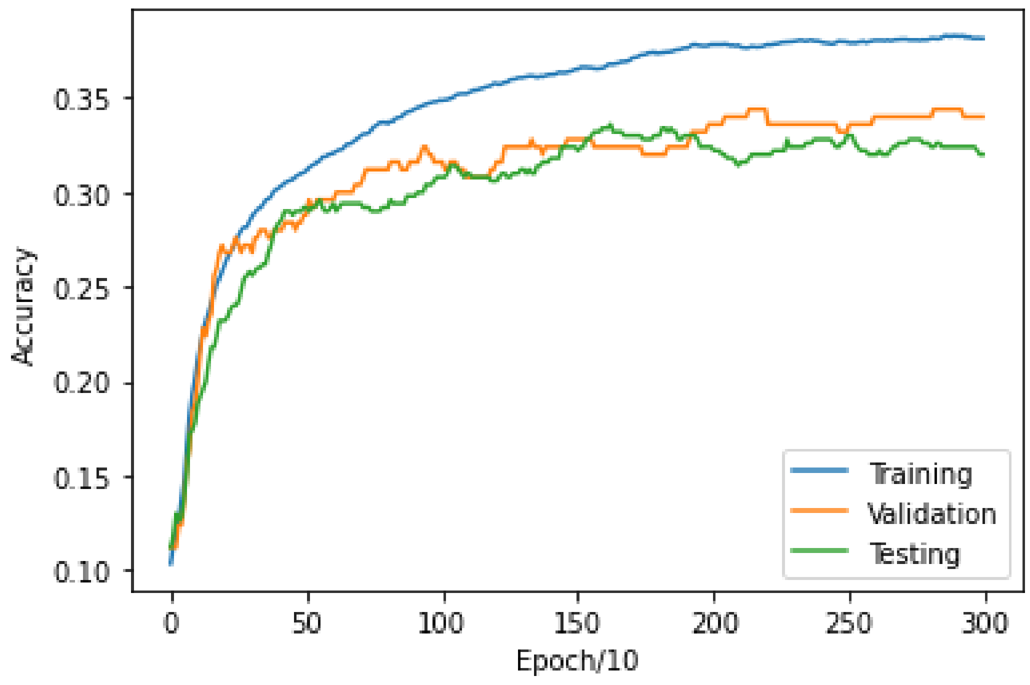
Inline Question 2. Discuss underfitting and overfitting as observed in the 5 graphs obtained by changing the regularization. Which weight_decay term gave you the best classifier performance? HINT: Do not just think in terms of best training set performance, keep in mind that the real utility of a machine learning model is when it performs well on data it has never seen before

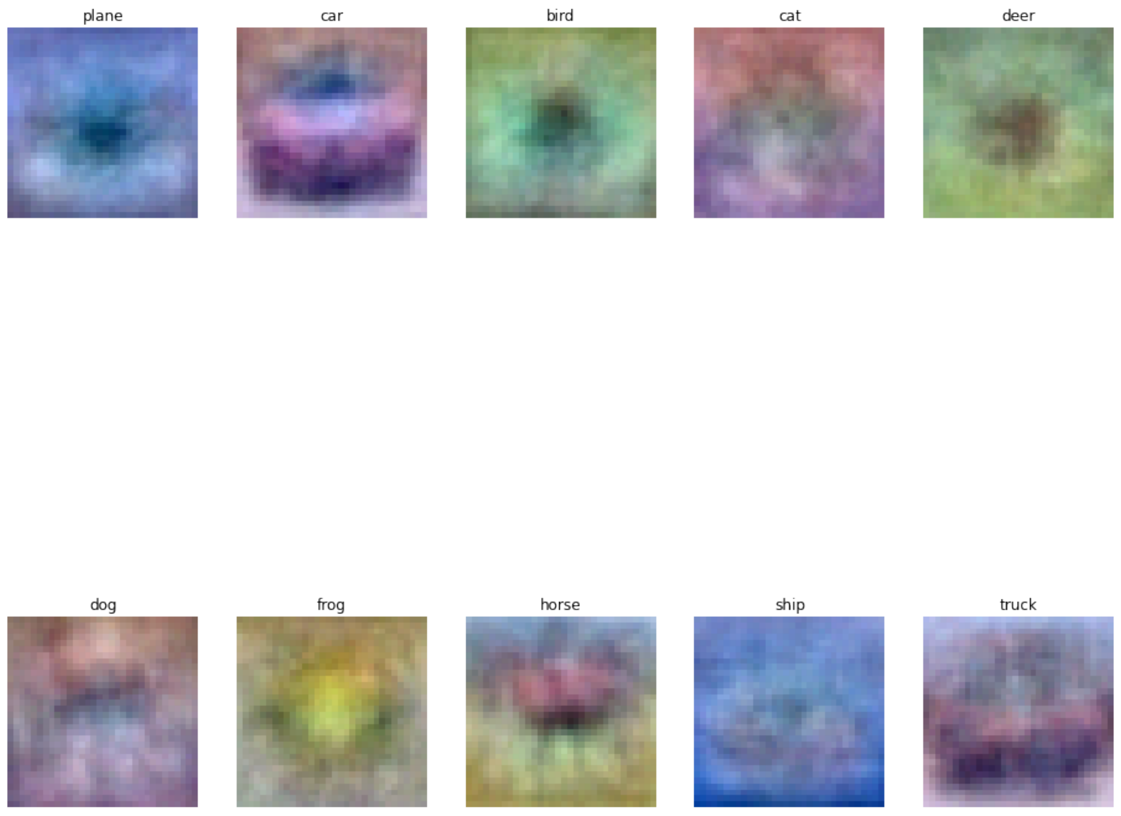
Your Answer:

```
[14]: #Overfitting occurs in weight_decay=0 as the difference between test accuracy␣
 ↪and training accuracy is bigger than the
#difference when weight_decay=0.1 and 1. Underfitting occurs when␣
 ↪weight_decay=100 as the test accuracy is much lower than the
#training accuracy by around 2.1%. I would use a weight_decay of 1 as it has␣
 ↪the best test accuracy out of all of them.
```

2.0.4 Visualize the filters (10%)

```
[12]: # These visualizations will only somewhat make sense if your learning rate and␣
      ↪weight_decay parameters were
      # properly chosen in the model. Do your best.
      #below is my code to choose best 3 parameters by looking at the graphs of lr=0.
      ↪01 and weight_decay=1 and num_epochs_total=3000
      learning_rate = 0.01 # You will be later asked to experiment with different␣
      ↪learning rates and report results
      num_epochs_total = 3000 # Total number of epochs to train the classifier
      #for num_epochs_total I changed it to 3000 as thats when the test accuracy␣
      ↪converged after running it with epochs=10000
      epochs_per_evaluation = 10 # Epochs per step of evaluation; We will evaluate␣
      ↪our model regularly during training
      N, D = dataset['x_train'].shape # Get training data shape, N: Number of␣
      ↪examples, D:Dimensionality of the data
      weight_decay = 1
      t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
      plot_accuracies(t_ac, v_ac, te_ac) # to see the accuracy with best results
      w = weights[:, :-1]
      w = w.reshape(10, 3, 32, 32).transpose(0, 2, 3, 1)

      w_min, w_max = np.min(w), np.max(w)

      fig=plt.figure(figsize=(16, 16))
      classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
      ↪'ship', 'truck']
      for i in range(10):
          fig.add_subplot(2, 5, i + 1)

          # Rescale the weights to be between 0 and 255
          wimg = 255.0 * (w[i, :, :, :].squeeze() - w_min) / (w_max - w_min)
          plt.imshow(wimg.astype(int))
          plt.axis('off')
          plt.title(classes[i])
      plt.show()
```


Accuracy
0.10
0.15
0.20
0.25
0.30
0.35
0
50
100
150
200
250
300
Epoch/10
Training
Validation
Testing


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

Inline Question 3. (10%)

a. Compare and contrast the performance of the 2 classifiers i.e. Linear Regression and Logistic Regression.
b. Which classifier would you deploy for your multiclass classification project and why?

Your Answer:

[16]:
```
#a. The linear regression is a little better than the logistic regression in␣
↪terms of test accuracy by about 3% which is what we want the machine␣
↪learning model to be the best at. The test accuracy also converges faster,␣
↪and both don't seem to be too affected by changes in weight decay unless the␣
↪weight decay is very big like 100. The learning rate affects both a lot␣
↪though.
#b. I would probably choose linear as I get a little higher test and training␣
↪accuracy compared to logistic and it also
#converges 1000 epochs faster.
```