ECE 285 Assignment 1: Logistic Regression

For this part of assignment, you are tasked to implement a logistic 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.

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

In [1]:

```python
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from ece285.utils.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,)
```

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

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.

In [7]:
```
# Import the algorithm implementation (TODO: Complete the Logistic Regression in algorithm
from ece285.algorithms import Logistic
from ece285.utils.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 rate
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 re
N, D = dataset[
    "x_train"
].shape  # Get training data shape, N: Number of examples, D:Dimensionality of the data
weight_decay = 0.00002

x_train = dataset["x_train"].copy()
y_train = dataset["y_train"].copy()
x_val = dataset["x_val"].copy()
y_val = dataset["y_val"].copy()
x_test = dataset["x_test"].copy()
y_test = dataset["y_test"].copy()
```

In [11]:
```
# Insert additional scalar term 1 in the samples to account for the bias as discussed in c
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)
```

In [4]:
```
# 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
```

In [5]:

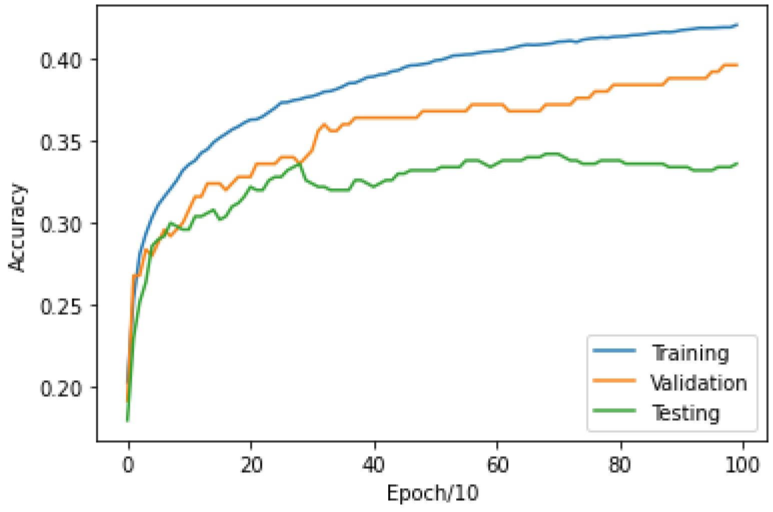
```
import matplotlib.pyplot as plt


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()
```

In [12]:

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

In [13]:

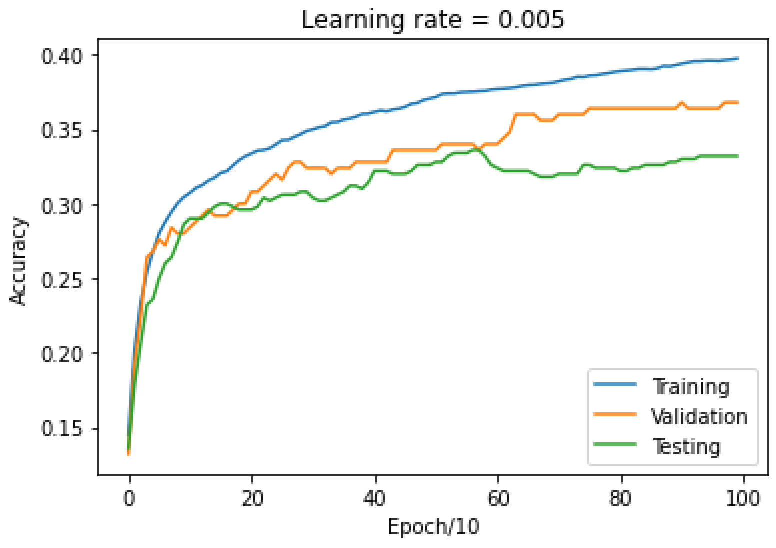
```
plot_accuracies(t_ac, v_ac, te_ac)
```

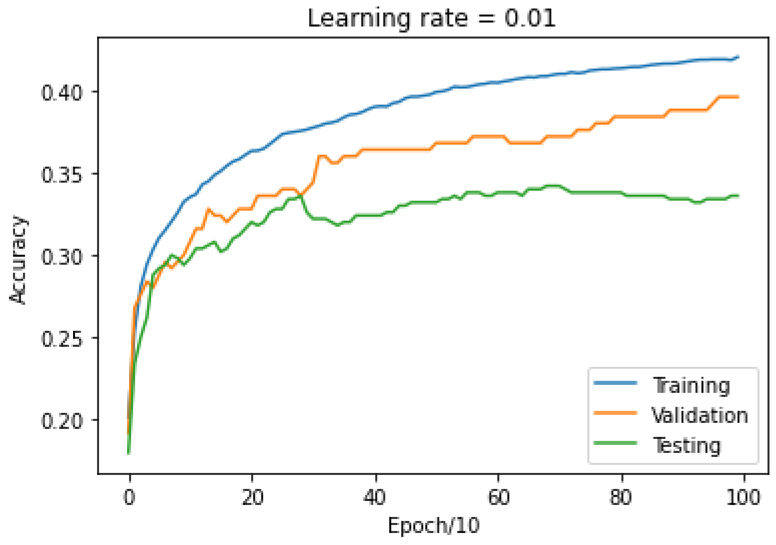


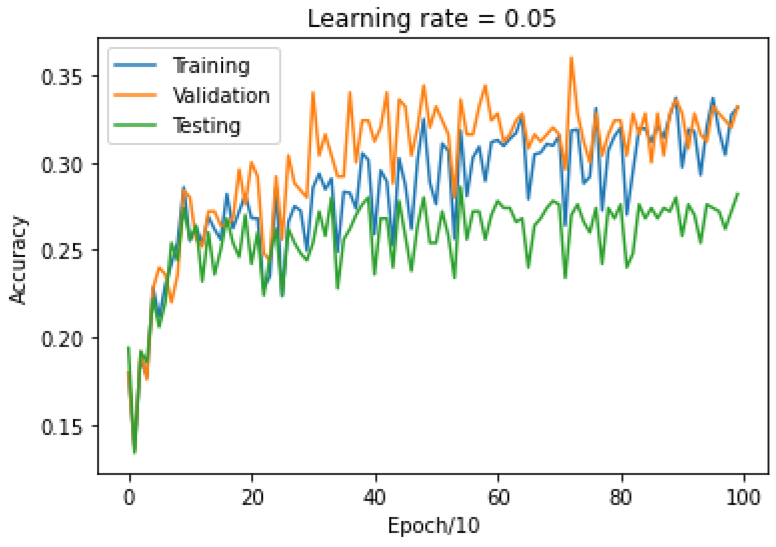
Try different learning rates and plot graphs for all (20%)

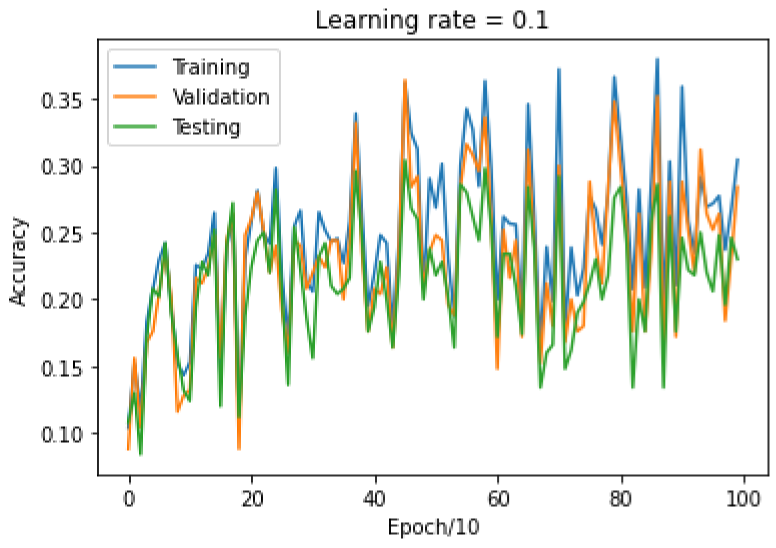
In [16]:

```
# TODO
# Repeat the above training and evaluation steps for the following learning rates and plot
# You need to submit all 5 graphs along with this notebook pdf
learning_rates = [0.005, 0.01, 0.05, 0.1]
weight_decay = 0.0  # No regularization for now

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

for lr in learning_rates:
    train_accu, val_accu, test_accu, weights = train(lr, weight_decay)
    plt.title('Learning rate = {}'.format(lr))
    plot_accuracies(train_accu, val_accu, test_accu)
```

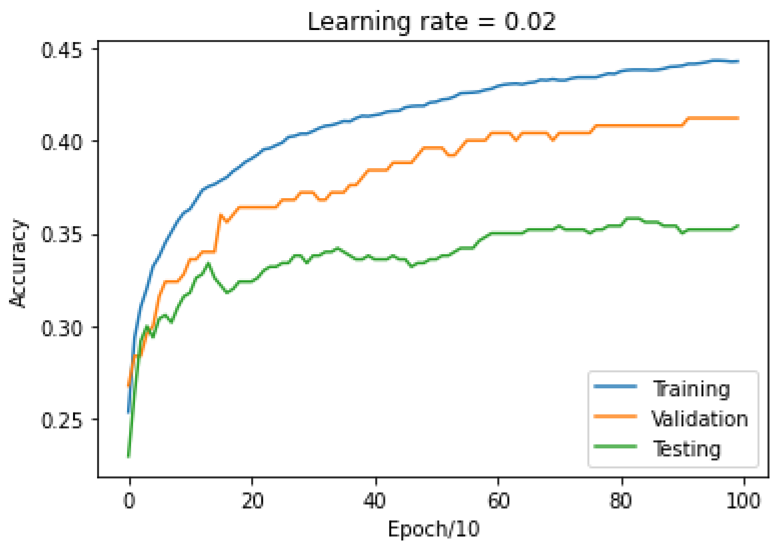

Learning rate = 0.005
Accuracy
Epoch/10
Training
Validation
Testing

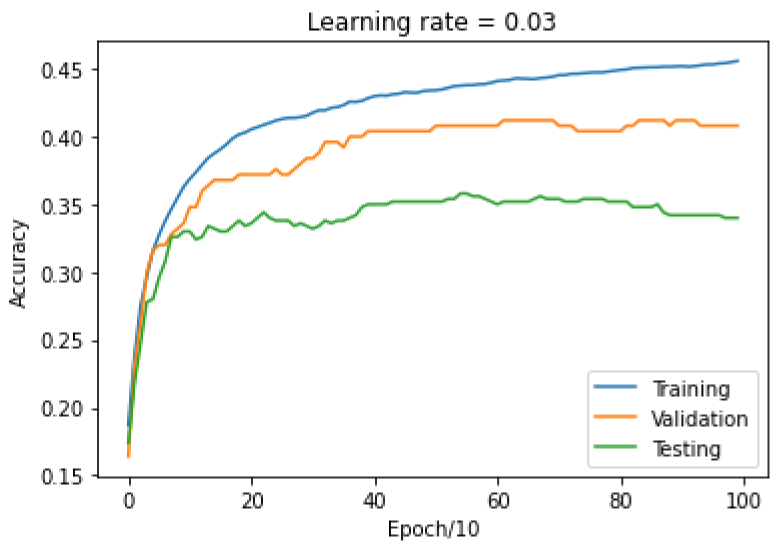

Learning rate = 0.01
Accuracy
Epoch/10
Training
Validation
Testing

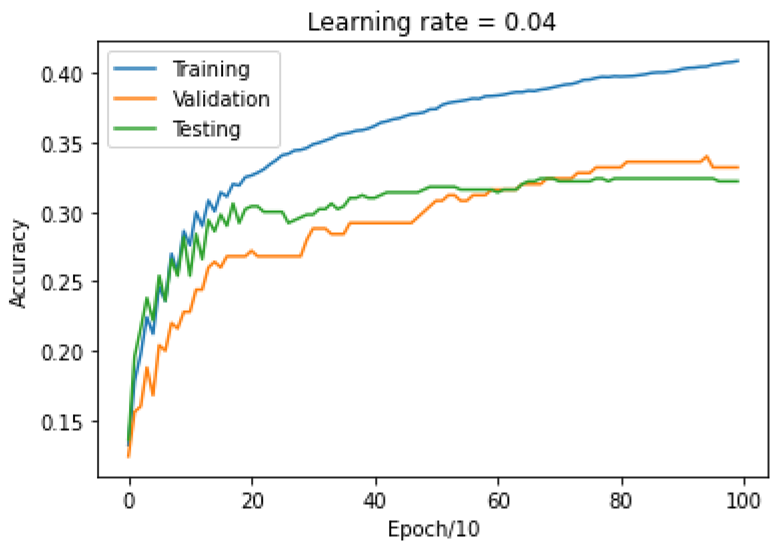

Learning rate = 0.05
Accuracy
Epoch/10
Training
Validation
Testing



In [17]:
```python
#my own try
learning_rates = [0.02, 0.03, 0.04]

for lr in learning_rates:
    train_accu, val_accu, test_accu, weights = train(lr, weight_decay)
    plt.title('Learning rate = {}'.format(lr))
    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.

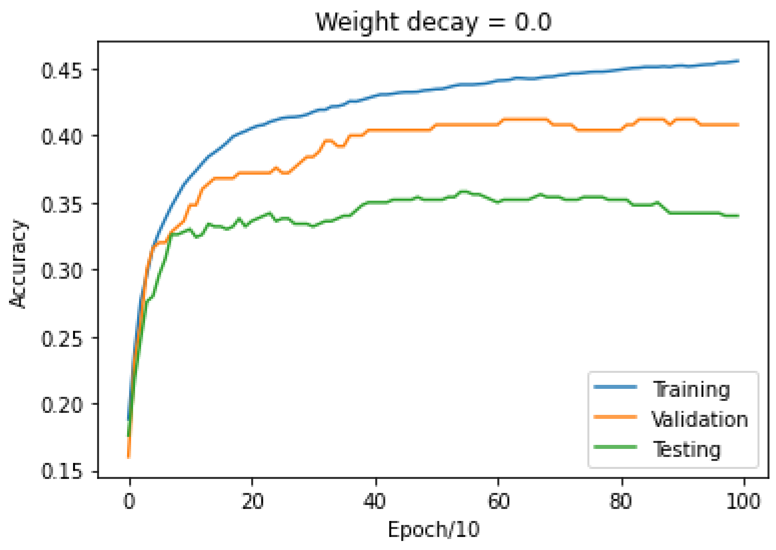
Your Answer:

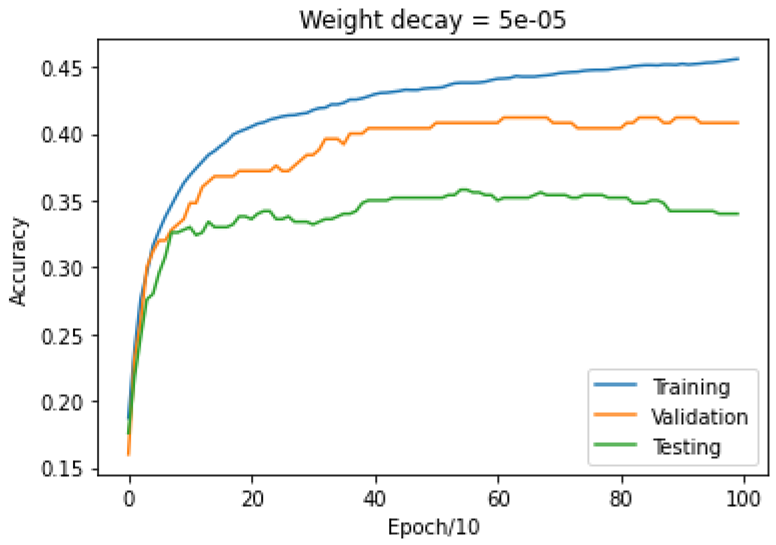
Form the graphs above, the best learning rate is 0.03. First, it does not make the model diverge. Second, it has a good learning rate, which reaches highest training accuracy (around 45%) and testing accuracy(around 35%) in 1000 epoches.

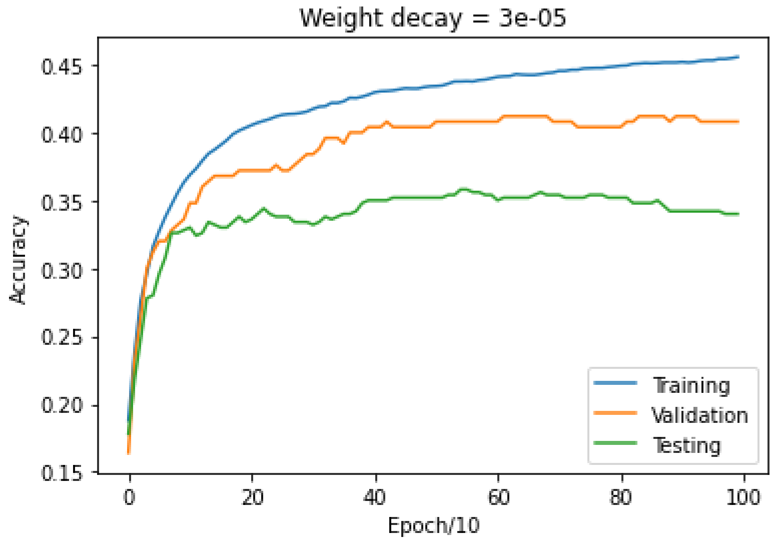
Regularization: Try different weight decay and plots graphs for all (20%)

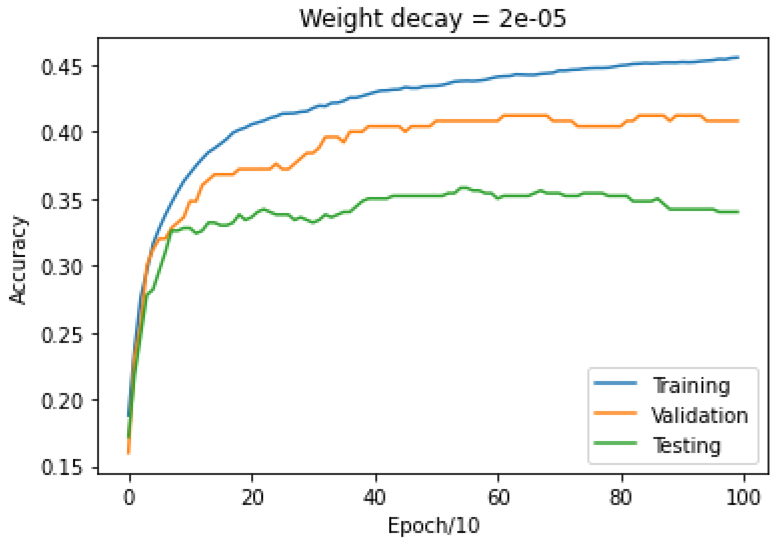
In [20]:

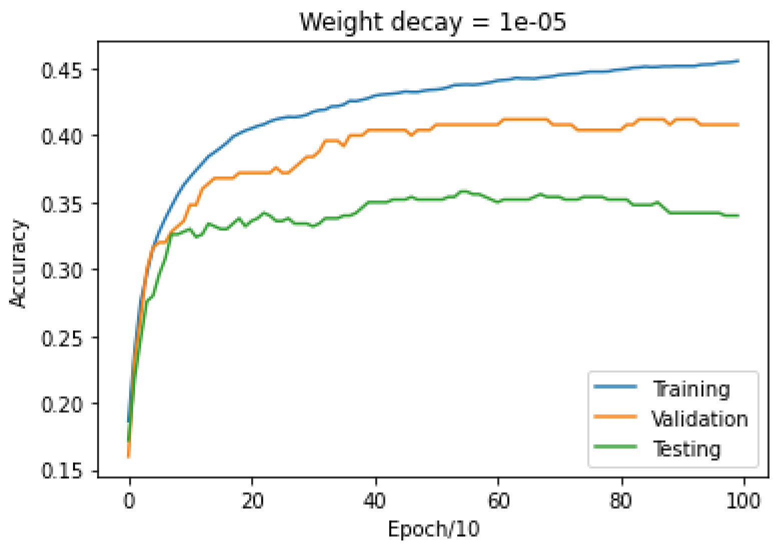
```
# Initialize a non-zero weight_decay (Regulzarization constant) term and repeat the traini
# Use the best learning rate as obtained from the above excercise, best_lr
weight_decays = [0.0, 0.00005, 0.00003, 0.00002, 0.00001, 0.000005]

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY ACHIEVE A BETTER
best_lr = 0.03
for weight_decay in weight_decays:
    train_accu, val_accu, test_accu,weight = train(best_lr, weight_decay)
    plt.title('Weight decay = {}'.format(weight_decay))
    plot_accuracies(train_accu, val_accu, test_accu)
```

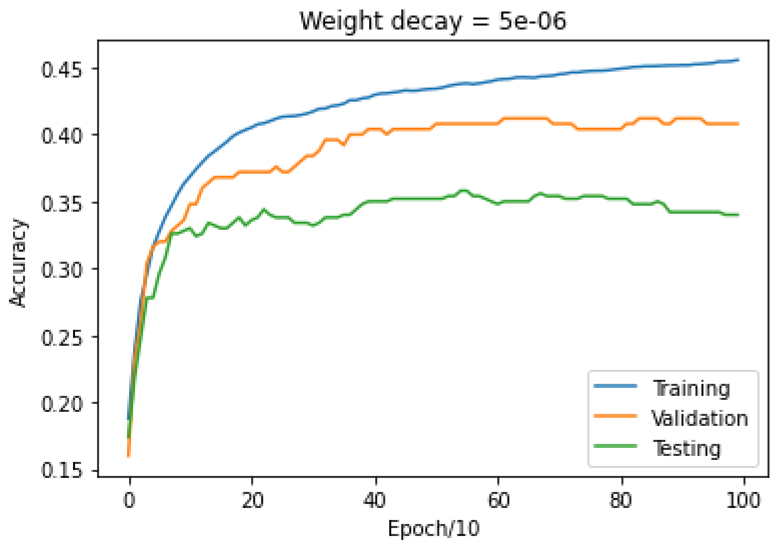



Weight decay = 5e-05
Accuracy
Epoch/10
Training
Validation
Testing

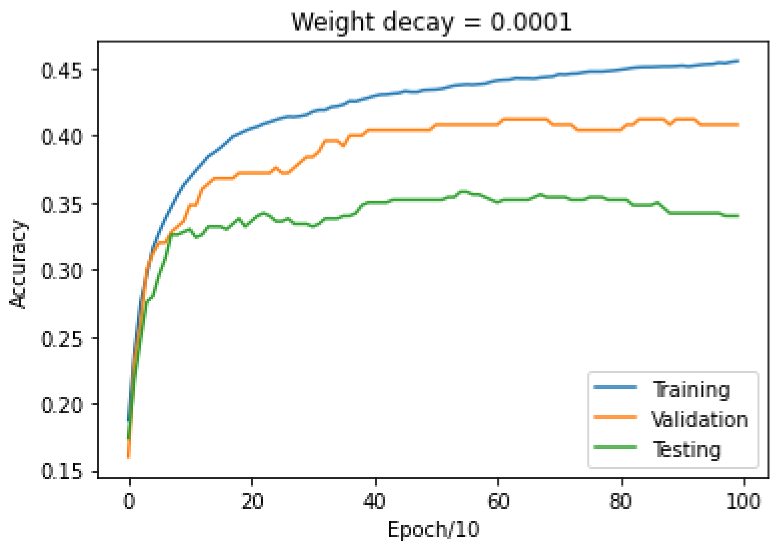

Weight decay = 3e-05
Accuracy
Epoch/10
Training
Validation
Testing

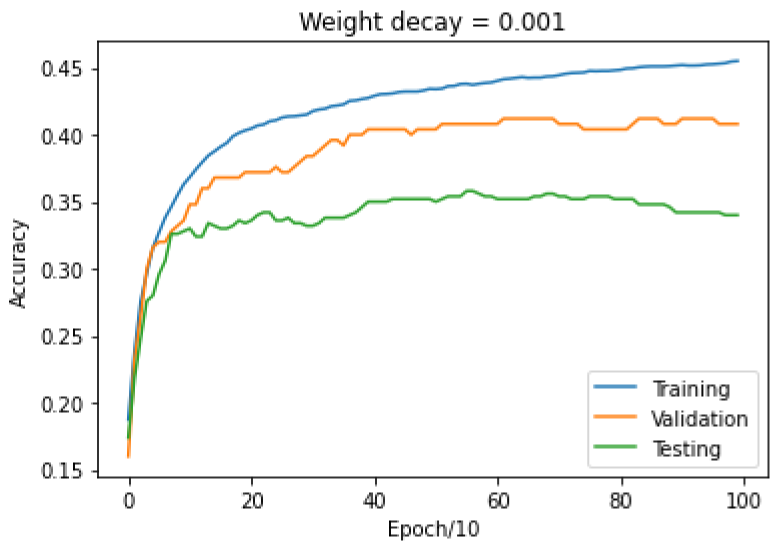

Weight decay = 2e-05
Accuracy
Epoch/10
Training
Validation
Testing

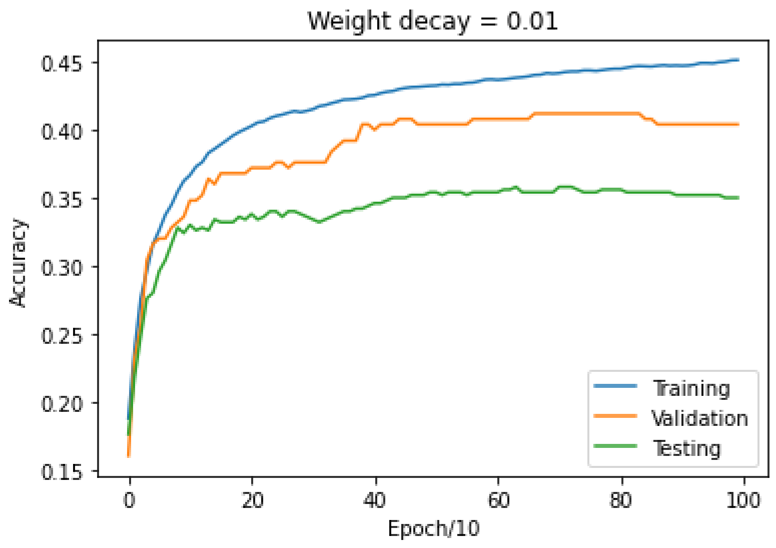


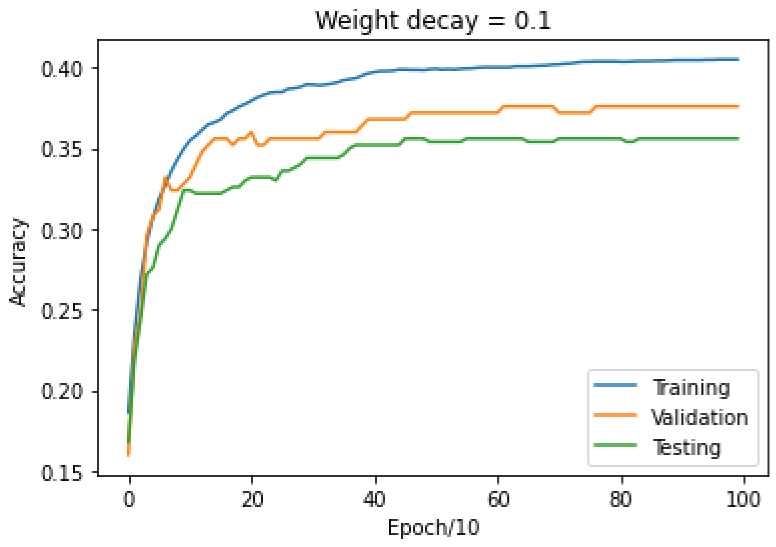


```
In [21]:
# My own try
weight_decays = [ 0.0001,0.001,0.01,0.1]

best_lr = 0.03
for weight_decay in weight_decays:
    train_accu, val_accu, test_accu,weight = train(best_lr, weight_decay)
    plt.title('Weight decay = {}'.format(weight_decay))
    plot_accuracies(train_accu, val_accu, test_accu)
```









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

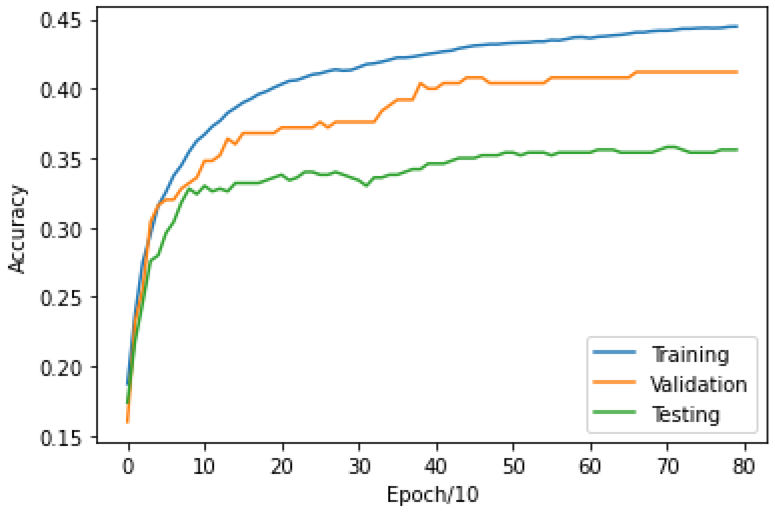
Your Answer:

All these graphs have a higher accuracy in training set than the test set, and as the training accuracy increases, the testing accuracy decreases after a certain point. This shows that there is some overfitting. I think the best weight decay is 0.01, because it has better validation accuracy and testing accuracy. The the validation accurancy and testing accuracy does not drop as training goes on, which means there's no overfitting.

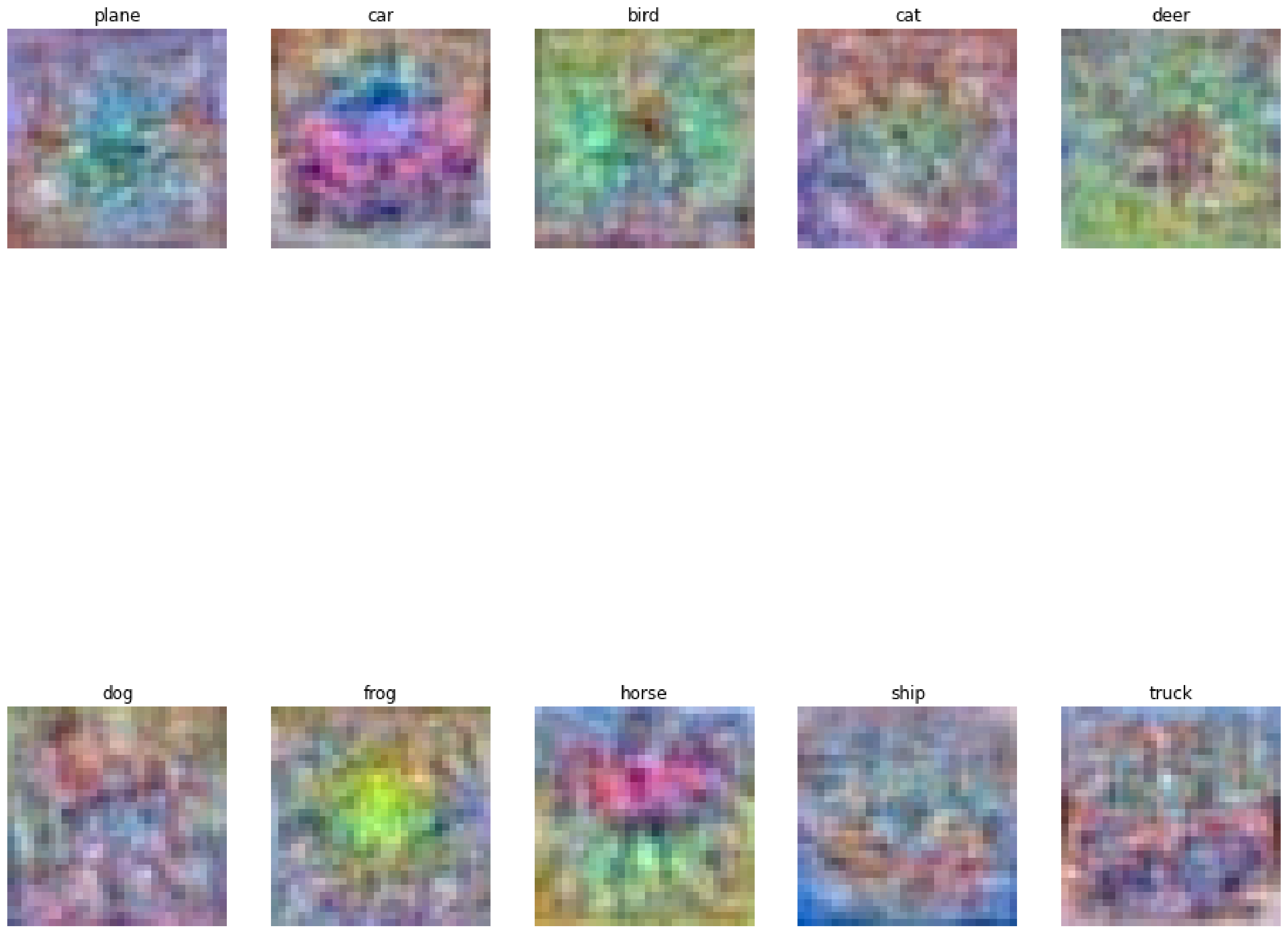
Visualize the filters (10%)

Under learning rate = 0.03 and weight decay = 0.01, the validation accurancy and testing accuracy does not drop much as training. The highest accuracy appears around epoch 800. Therefore I use epoch =800 as the hyperparameter

In [22]:
```
learning_rate = 0.03  # You will be later asked to experiment with different learning rate
num_epochs_total = 800  # Total number of epochs to train the classifier
epochs_per_evaluation = 10  # Epochs per step of evaluation; We will evaluate our model re
weight_decay = 0.01

train_accu, val_accu, test_accu,weight = train(learning_rate, weight_decay)

plot_accuracies(train_accu, val_accu, test_accu)
```



In [23]:
```
# These visualizations will only somewhat make sense if your learning rate and weight_deca
# properly chosen in the model. Do your best.

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()
```



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:

a. These two model have similiar performance. Both can reach a training accuracy around 45% and testing accuracy around 35%. However, the training model accuracy drops after a certain epoch while the logistics model does not, this shows that linear regression model can lead to overfit.

b. I will choose Logistics regression for multiclass classification because it's more robust and more approiate for the problem. The logistic model is more suitable for classification (only produce answer between 0 and 1) while the linear model can produce probability larger than 1 or smaller than 0. Besides, the sigmoid function creates a larger boundary (distance to hyperplane) than the linear model.