ECE 285 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.

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

In [22]:
```
# Import Packages
import numpy as np
import matplotlib.pyplot as plt
```

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

In [23]:
```
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)
```

```
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,)
```

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 Eucliean distance between samples.

In [24]:
```
from ece285.algorithms import KNN

knn = KNN(num_class=10)
knn.train(
```

```
    x_train=dataset["x_train"],
    y_train=dataset["y_train"],
    k=5,
)
```

Compare the time consumption of different method

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

In [25]:
```
from ece285.utils.evaluation import get_classification_accuracy
```

Two Loop Version:

In [26]:
```
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: 761.9649147987366
Test Accuracy: 0.278
```

One Loop Version

In [27]:
```
import time

c_t = time.time()
prediction = knn.predict(dataset["x_test"], 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: 28.30700421333313
Test Accuracy: 0.278
```

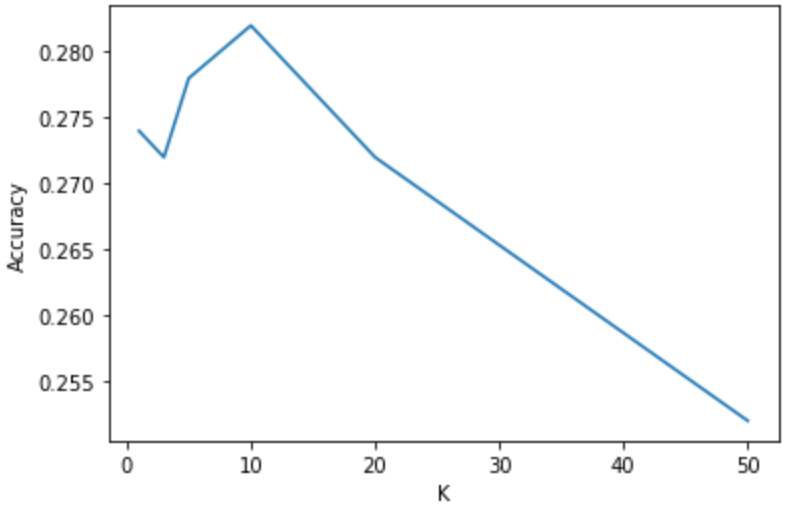
Your different implementation should output the exact same result

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.

In [32]:
```
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()

print(accuracies)
```



Inline Question 1:

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

Your Answer:

First, the overall accuracy of KNN prediction is low, around 30%, the highest accuracy is at k=10.

Second, the 1 loop versionis much more faster that the two loop version. The one loop version takes around 25 seconds. The 2 loop version takes about 12 minutes (Maybe I have the wrong implementation).

Last but not least, the accuracy does not monotonically increase with K. It reaches a peak at k=10 (28.2%) and start decreasing. This confirms that KNN algorithm does not necessarily have better performance with larger K. The result may be infected by the proportion of each class.

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.

In [29]:

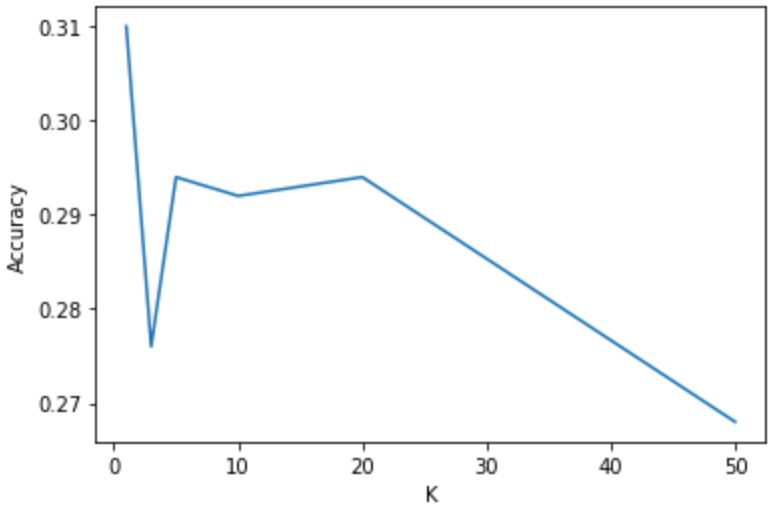
```
from ece285.utils.data_processing import get_cifar10_data
from ece285.utils.data_processing import HOG_preprocess
from functools import partial

# Delete previous dataset to save memory
del dataset
del knn

# 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: 8.607496500015259
```

In [30]:

```
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()
```



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.

Your Answer:

First, the overal accuracy of HOG representation is around 29%, which is higher than the pixel representation, but still in unsatisfying.

Second, the accuracy of algorithm varies with the K. The highest accuracy apperas at K=1 (31%), then it rise and fall around 29%. This also shows that the accuracy of KNN may not increase with K. The result can be affected by the proportion of the classes

ECE 285 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.

In [79]:
```
# 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 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,)
```

In [80]:
```
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"]
```

In [81]:
```
# 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
)
```

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.

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.

In [82]:
```
# Import the algorithm implementation (TODO: Complete the Linear Regression in algorithms/
from ece285.algorithms import Linear
from ece285.utils.evaluation import get_classification_accuracy

num_classes = 10  # Cifar10 dataset has 10 different classes

# Initialize hyper-parameters
learning_rate = 0.0001  # You will be later asked to experiment with different learning ra
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:Dimen
weight_decay = 0.0
```

In [83]:
```
# 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 [86]:
```
# 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
```

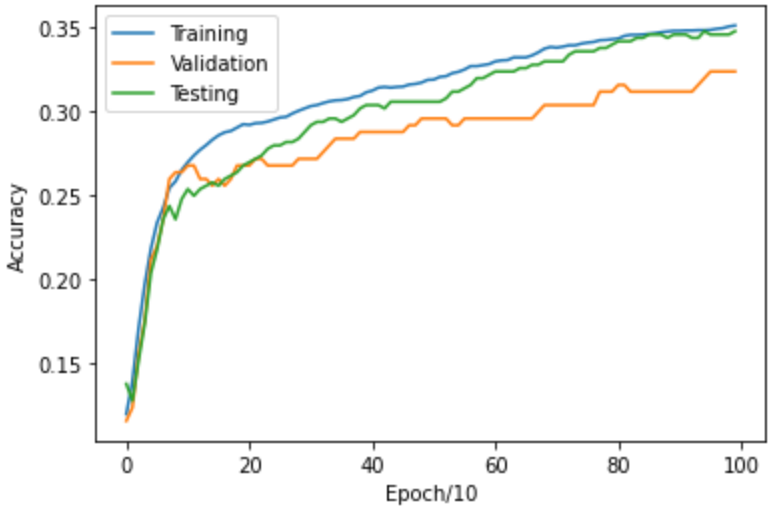
Plot the Accuracies vs epoch graphs

In [85]:
```
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 [87]:
```
# Run training and plotting for default parameter values as mentioned above
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
```

In [88]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```



In [89]:
```
#check the ground truth
from sklearn.linear_model import LinearRegression

y_encode = np.zeros((N,10))
for i in range(10):
    a = np.array(y_train==i)
    y_encode[:,i] = 2*a-1 #{-1,1} encoding

predict_y = np.zeros((500,10))

for i in range(10):
    reg = LinearRegression()
    reg.fit(x_train,y_encode[:,i])
    predict_y[:,i] = reg.predict(x_test)

predict_label = np.argsort(predict_y,axis=1)[:,-1]
get_classification_accuracy(predict_label,y_test)
```

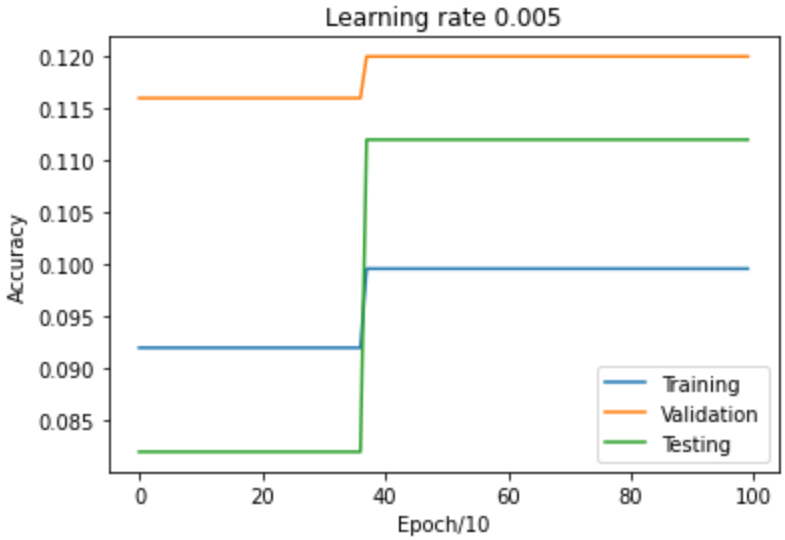
Out[89]:
```
0.204
```

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

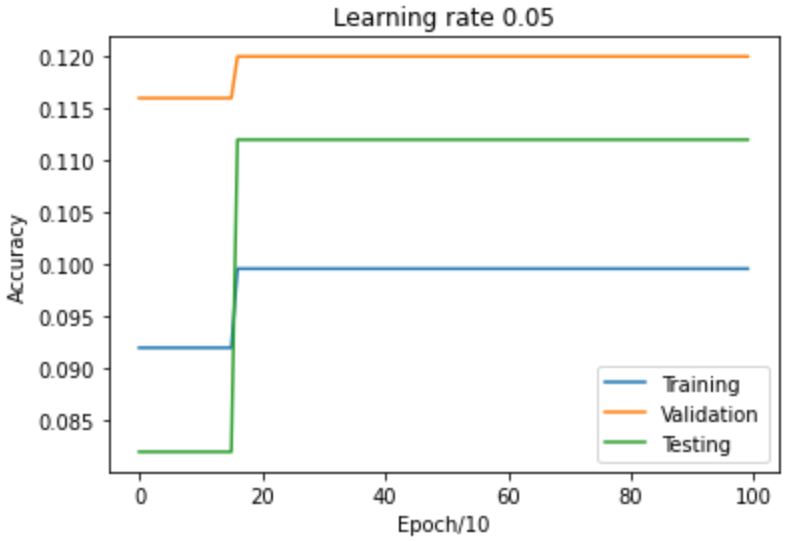
In [90]:
```
# TODO
# Repeat the above training and evaluation steps for the following learning rates and plot
# You need to submit all 5 graphs along with this notebook pdf
learning_rates = [0.005, 0.05, 0.1, 0.5, 1.0]
weight_decay = 0.0  # No regularization for now
```

```
for i in learning_rates:
    t_ac, v_ac, te_ac, weights = train(i, weight_decay)
    plt.title('Learning rate {}'.format(i))
    plot_accuracies(t_ac, v_ac, te_ac)


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

# 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)
```

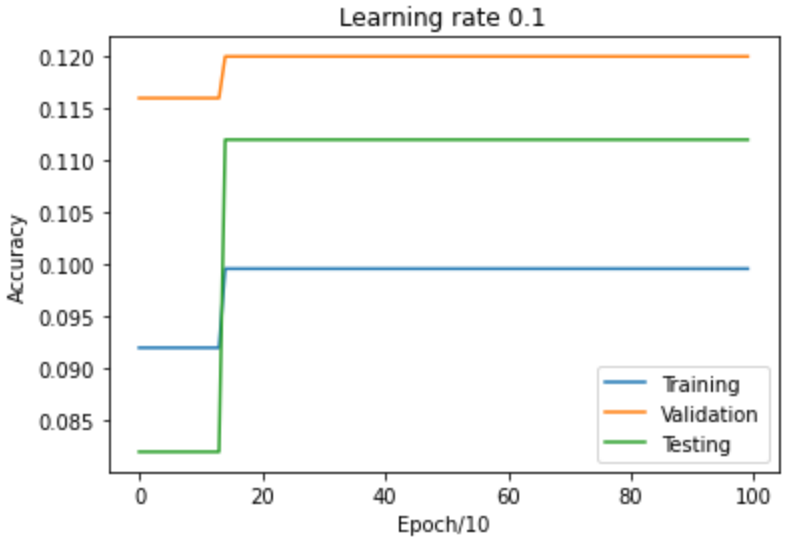
```
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:48: Runti
meWarning: overflow encountered in matmul
  grad = 2*X_train.T@(y_predict.T-y_encode)/N #DxN @ N*10
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:47: Runti
meWarning: invalid value encountered in matmul
  y_predict = self.w@X_train.T
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:49: Runti
meWarning: invalid value encountered in multiply
  self.w = self.w-(grad.T + self.weight_decay*self.w)*self.lr
```



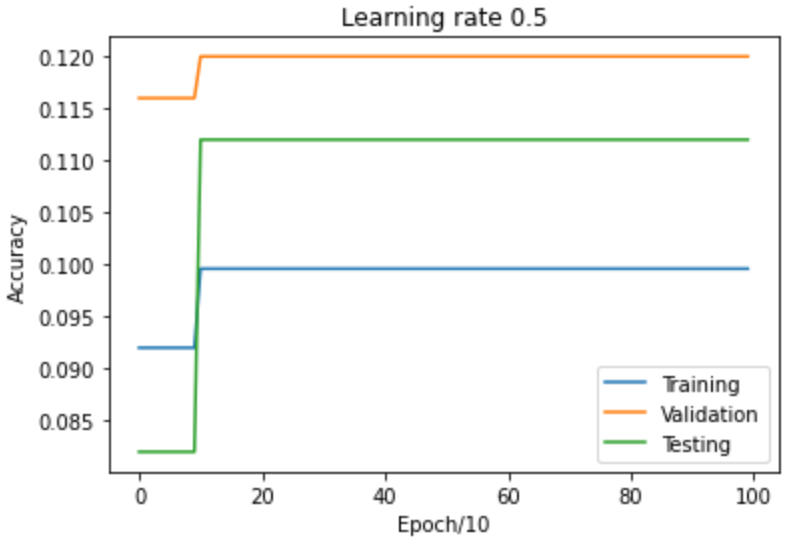

```
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:48: Runti
meWarning: overflow encountered in matmul
  grad = 2*X_train.T@(y_predict.T-y_encode)/N #DxN @ N*10
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:47: Runti
meWarning: invalid value encountered in matmul
  y_predict = self.w@X_train.T
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:49: Runti
meWarning: invalid value encountered in multiply
  self.w = self.w-(grad.T + self.weight_decay*self.w)*self.lr
```



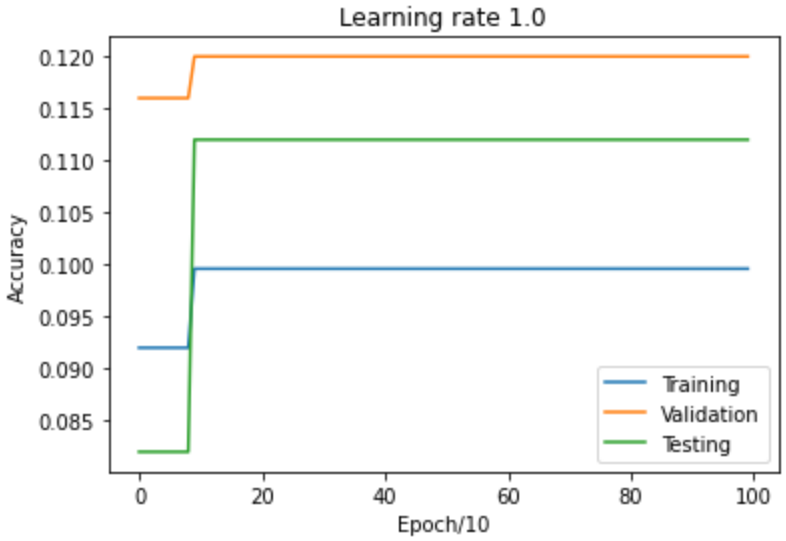

```
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:48: Runti
meWarning: overflow encountered in matmul
  grad = 2*X_train.T@(y_predict.T-y_encode)/N #DxN @ N*10
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:47: Runti
meWarning: invalid value encountered in matmul
  y_predict = self.w@X_train.T
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:49: Runti
meWarning: invalid value encountered in multiply
  self.w = self.w-(grad.T + self.weight_decay*self.w)*self.lr
```




```
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:48: Runti
meWarning: overflow encountered in matmul
  grad = 2*X_train.T@(y_predict.T-y_encode)/N #DxN @ N*10
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:47: Runti
meWarning: invalid value encountered in matmul
  y_predict = self.w@X_train.T
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:49: Runti
meWarning: invalid value encountered in multiply
  self.w = self.w-(grad.T + self.weight_decay*self.w)*self.lr
```

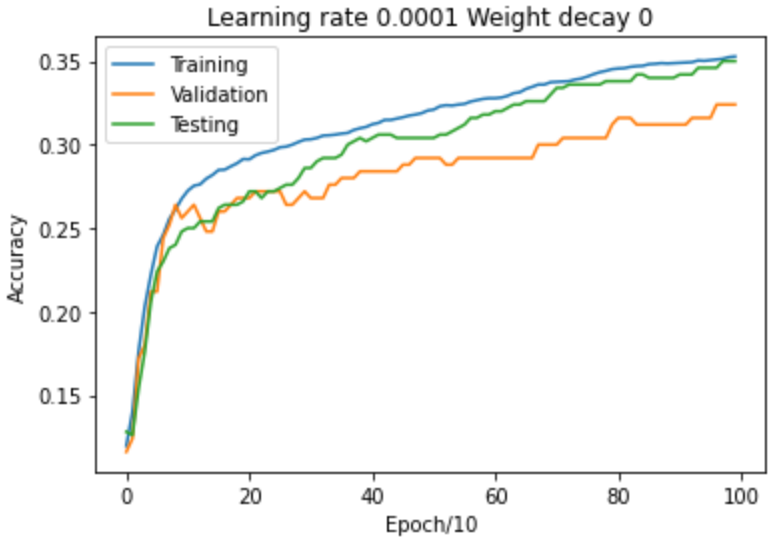


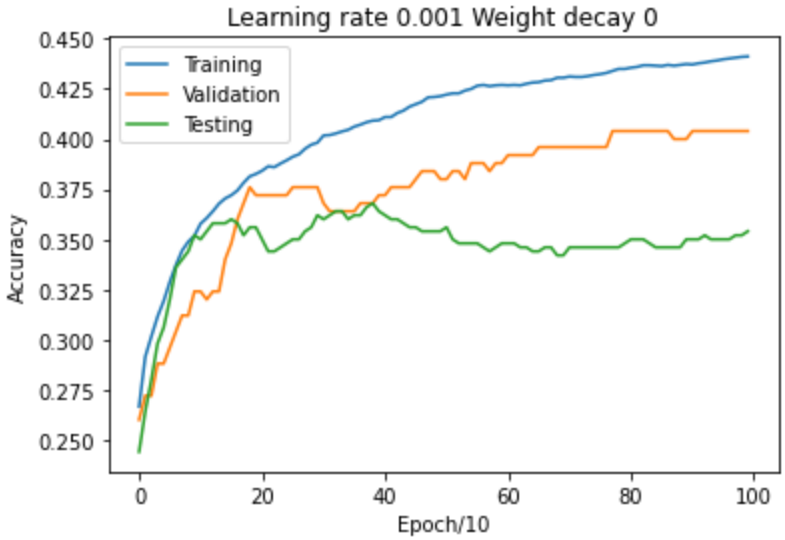

```
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:48: Runti
meWarning: overflow encountered in matmul
  grad = 2*X_train.T@(y_predict.T-y_encode)/N #DxN @ N*10
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:47: Runti
meWarning: invalid value encountered in matmul
  y_predict = self.w@X_train.T
c:\Users\lizhu\Desktop\ece285\assignment1\ece285\algorithms\linear_regression.py:49: Runti
meWarning: invalid value encountered in multiply
  self.w = self.w-(grad.T + self.weight_decay*self.w)*self.lr
```



In [64]:

```
#my own try
learning_rates = [0.0001,0.0005,0.001]
weight_decay=[0]
for i in learning_rates:
    for j in weight_decay:
        t_ac, v_ac, te_ac, weights = train(i, j)
        plt.title('Learning rate {} Weight decay {}'.format(i,j))
        plot_accuracies(t_ac, v_ac, te_ac)
```





Inline Question 1.

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

Your Answer:

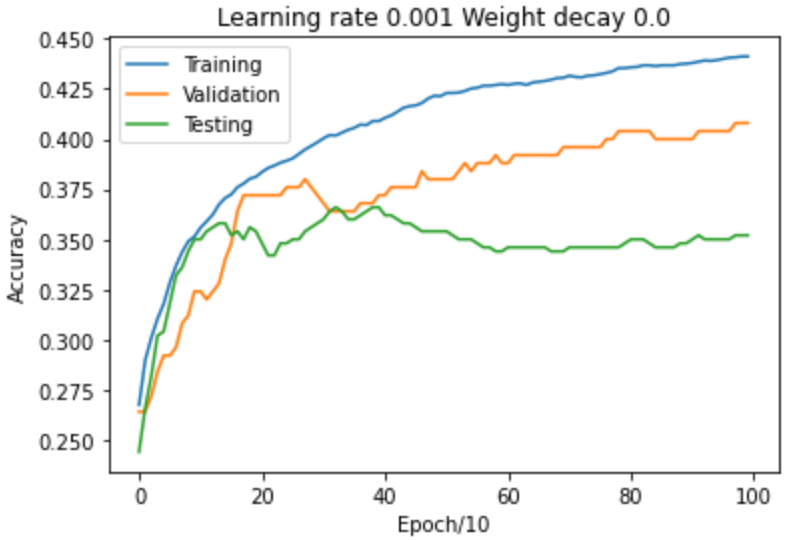
The best learning rate happens at 0.001. First, the learning rate larger than it cannot make the model converge, and doesnot help on training the data. Second, thr learning rate smaller than the model learns slower. In same number of epochs, leanrning rate = 0.001 achieves the best validation and test accuracy.

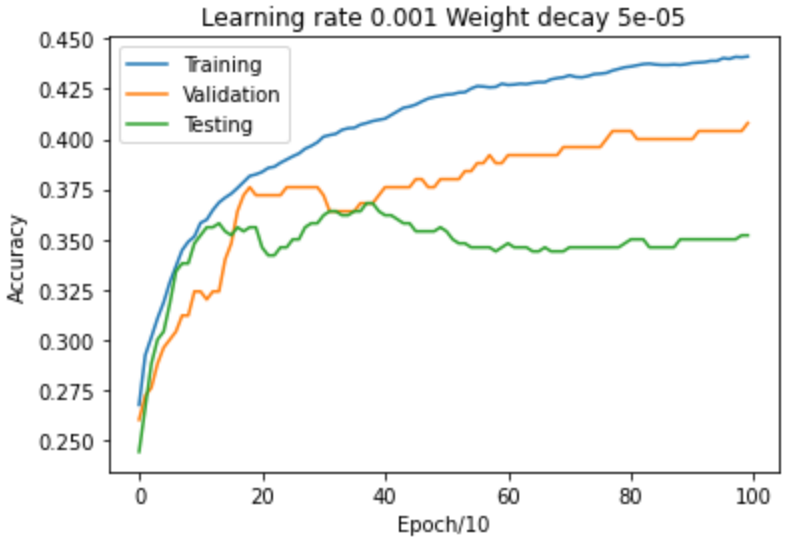
It can be seen that around 400 epochs the model begins to overfit. Therefore a regulzarization is needed.

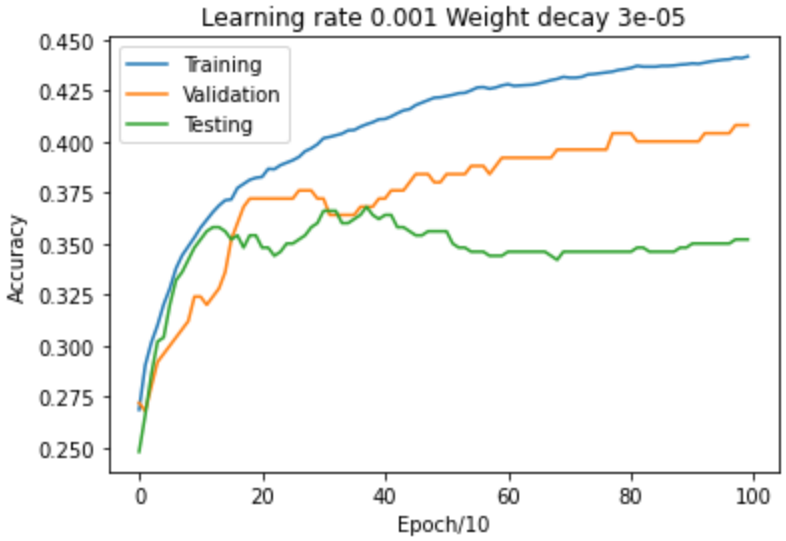
Regularization: Try different weight decay and plot graphs for all (20%)

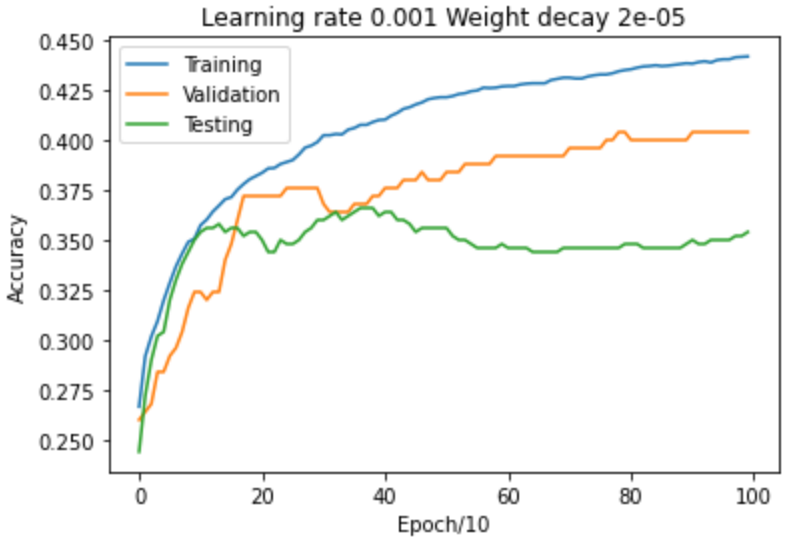
In [66]:

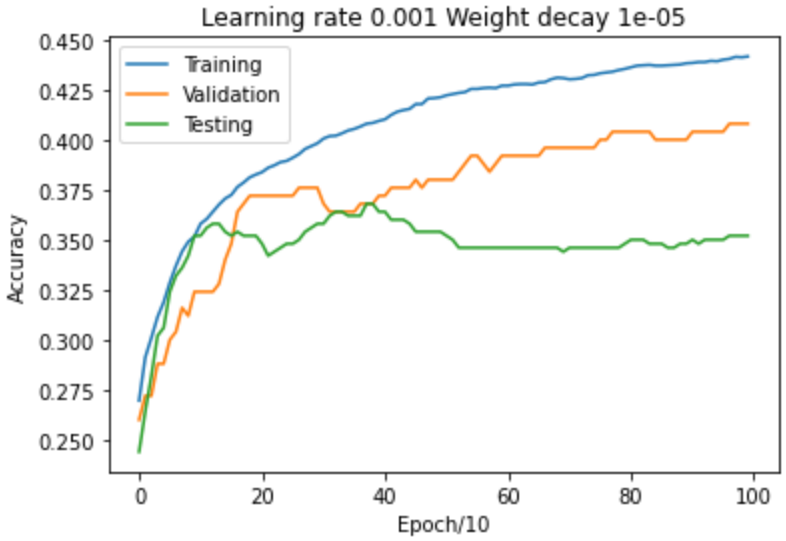
```
# Initialize a non-zero weight_decay (Regulzarization constant) term and repeat the traini
# Use the best learning rate as obtained from the above excercise, best_lr
weight_decays = [0.0, 0.00005, 0.00003, 0.00002, 0.00001, 0.000005]

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY ACHIEVE A BETTER
i = 0.001
for j in weight_decays:
    t_ac, v_ac, te_ac, weights = train(i, j)
    plt.title('Learning rate {} Weight decay {}'.format(i,j))
    plot_accuracies(t_ac, v_ac, te_ac)
```

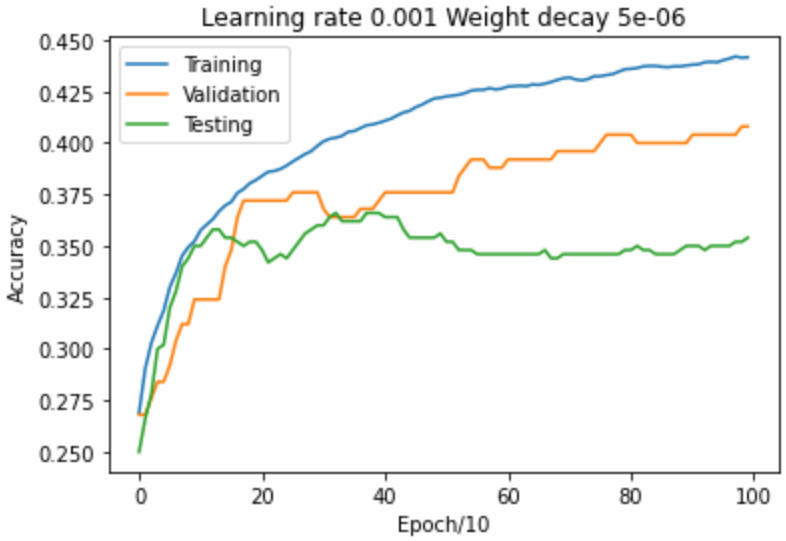





Learning rate 0.001 Weight decay 3e-05
Training
Validation
Testing
Accuracy
Epoch/10
0.450
0.425
0.400
0.375
0.350
0.325
0.300
0.275
0.250
0
20
40
60
80
100

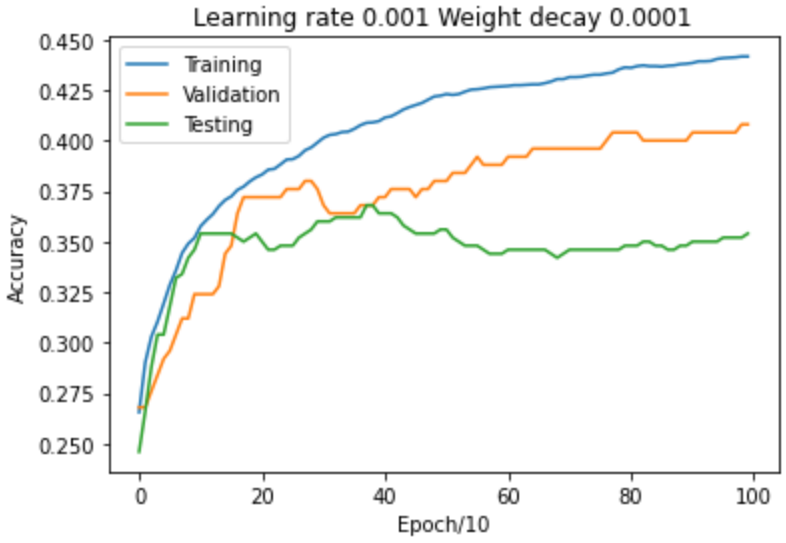

Learning rate 0.001 Weight decay 2e-05
Training
Validation
Testing
Accuracy
Epoch/10
0.450
0.425
0.400
0.375
0.350
0.325
0.300
0.275
0.250
0
20
40
60
80
100

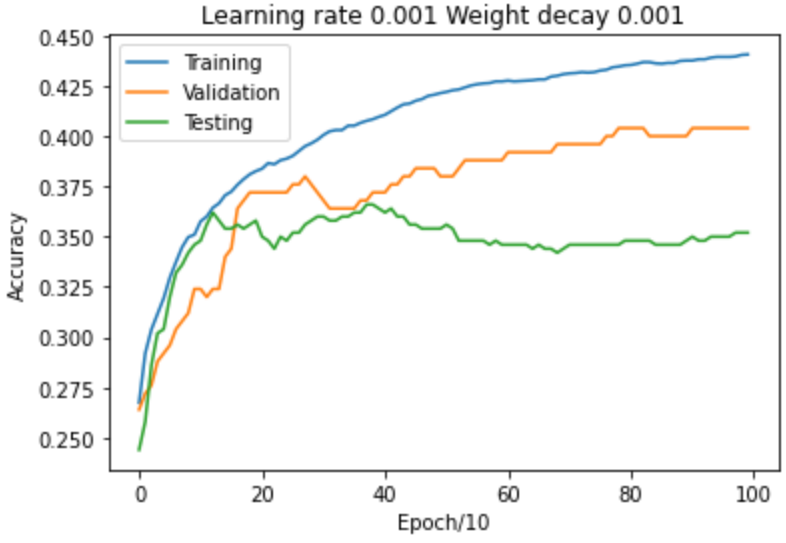

Learning rate 0.001 Weight decay 1e-05
Training
Validation
Testing
Accuracy
Epoch/10
0.450
0.425
0.400
0.375
0.350
0.325
0.300
0.275
0.250
0
20
40
60
80
100

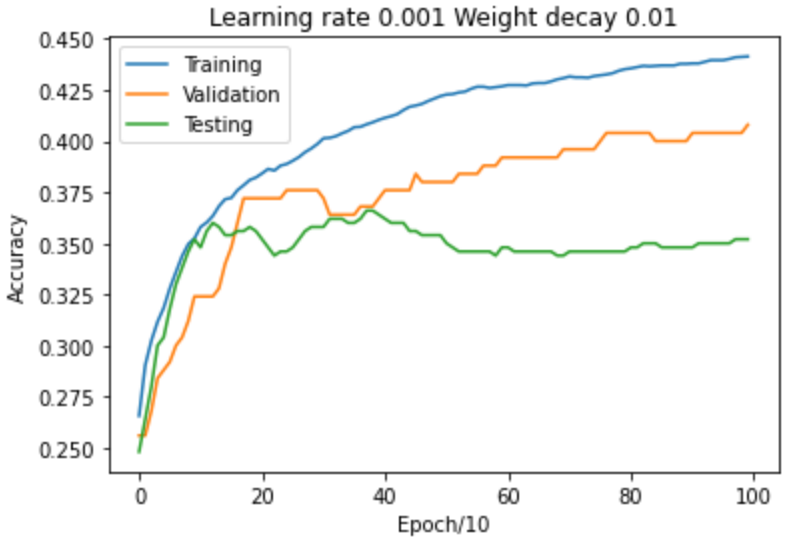


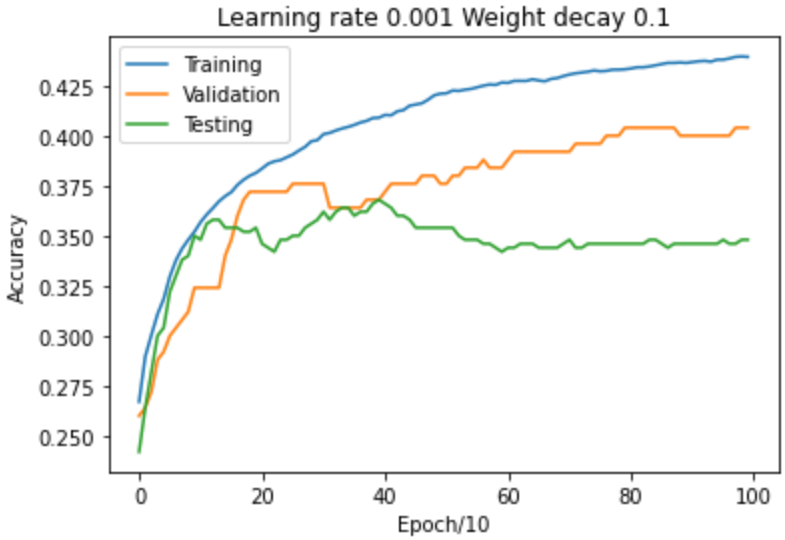
In [67]:

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

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY ACHIEVE A BETTER
i=0.001
for j in weight_decays:
    t_ac, v_ac, te_ac, weights = train(i, j)
    plt.title('Learning rate {} Weight decay {}'.format(i,j))
    plot_accuracies(t_ac, v_ac, te_ac)
```









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:

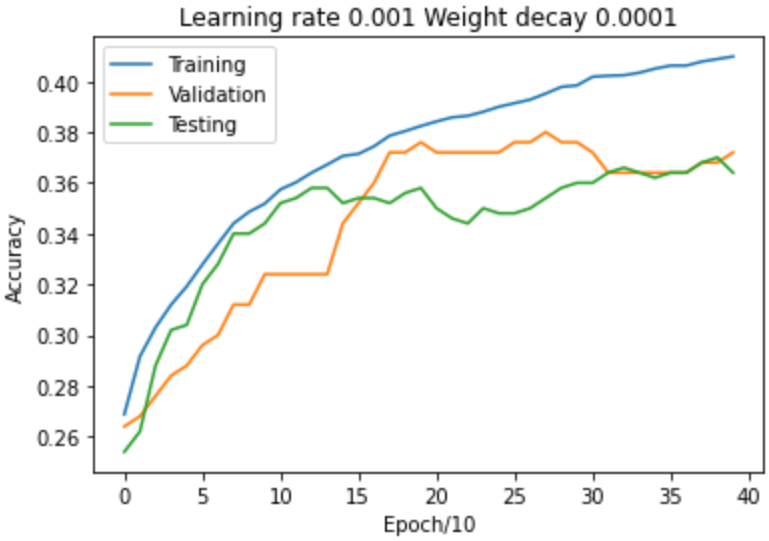
All these 5 graphs have a higher accuracy in training set than the test set, and as the training accuracy increases, the testing accuracy decreases after a certain point. This shows that there is some overfitting. I think the best weight decay is 0.0001, because it has better validation accuracy and testing accuracy.

Visualize the filters (10%)

The plot shows that the model is about to overfit around epoch 400. Therefore I choose epoch = 400 to train the model.

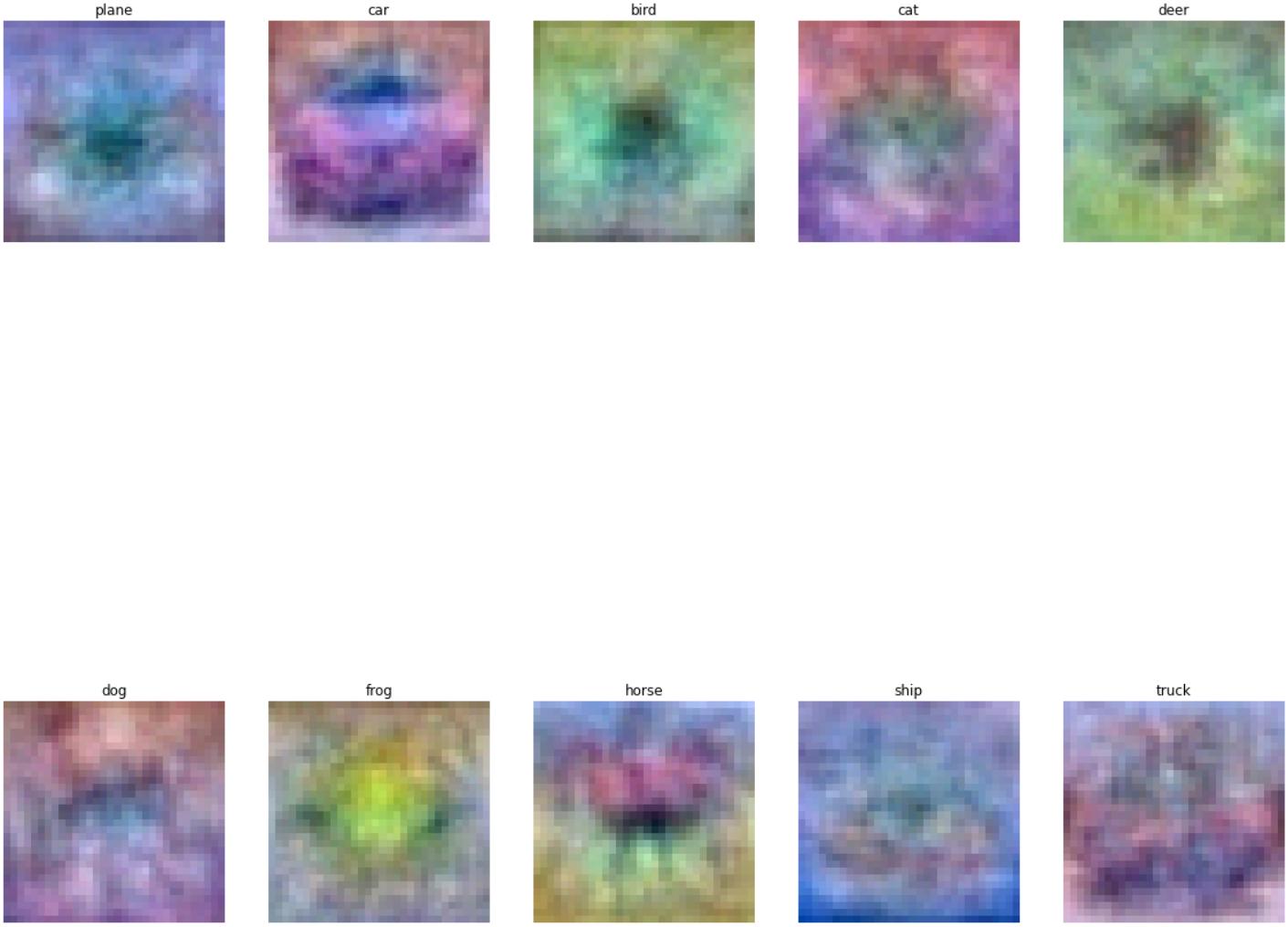
In [72]:

```
learning_rate = 0.001  # You will be later asked to experiment with different learning rat
num_epochs_total = 400  # 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 = 0.0001

t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
plt.title('Learning rate {} Weight decay {}'.format(learning_rate,weight_decay))
plot_accuracies(t_ac, v_ac, te_ac)
```



In [73]:

```
# 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=(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 obtai
# Report the 3 hyperparameters you used to obtain the best model.

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

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]:

```
# 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]:

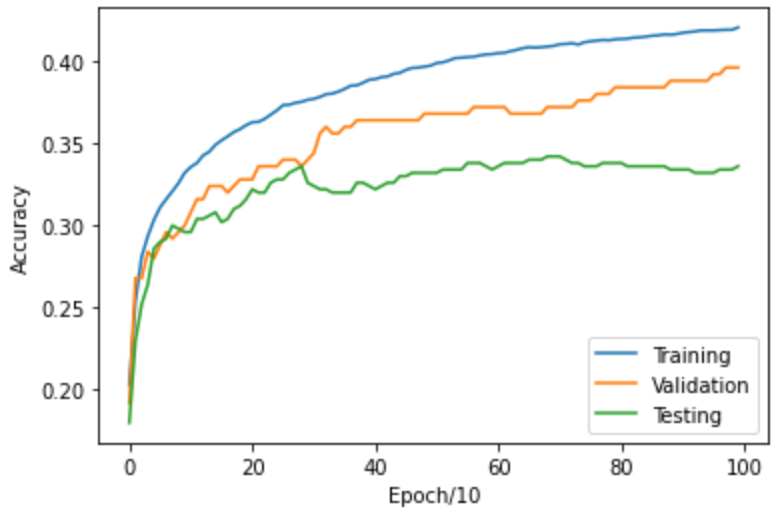
```
import matplotlib.pyplot as plt


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]:

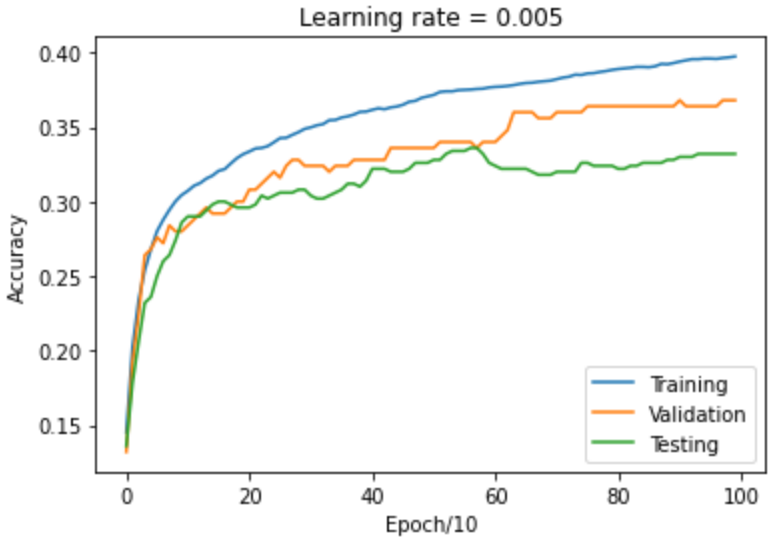
```
plot_accuracies(t_ac, v_ac, te_ac)
```

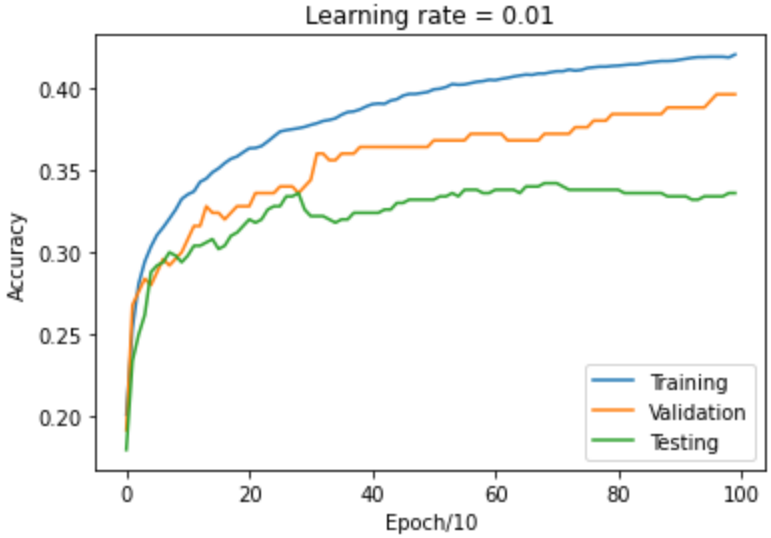


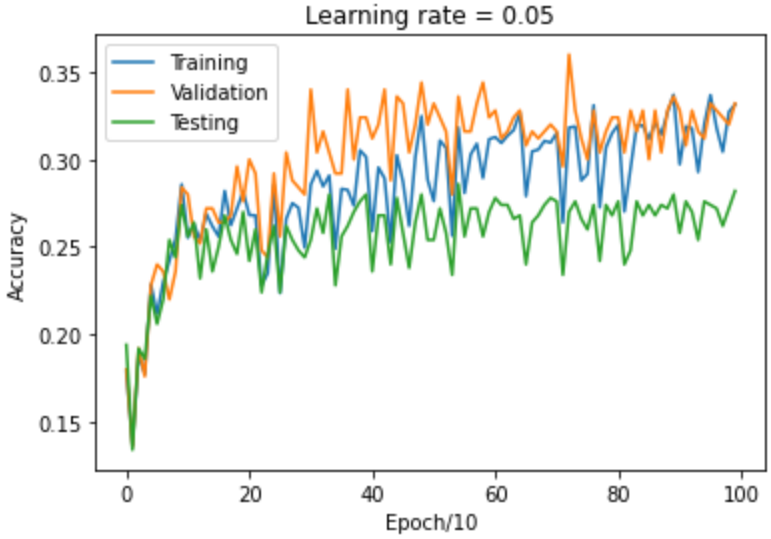
Try different learning rates and plot graphs for all (20%)

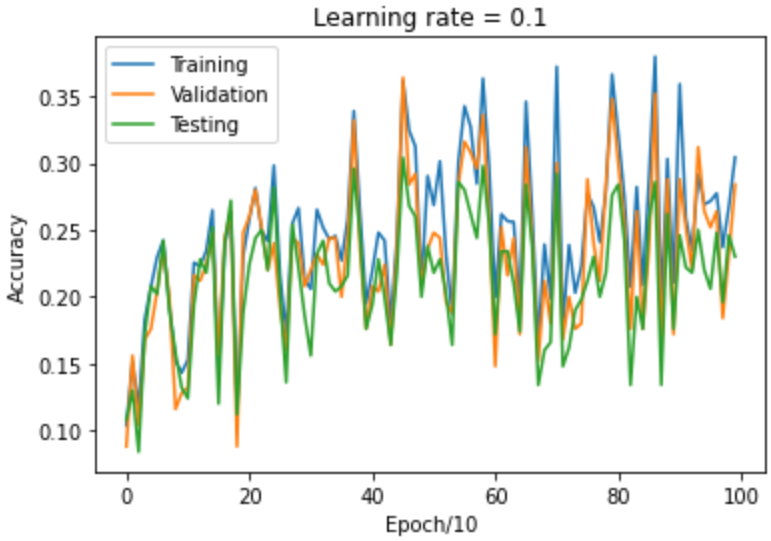
In [16]:

```
# 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

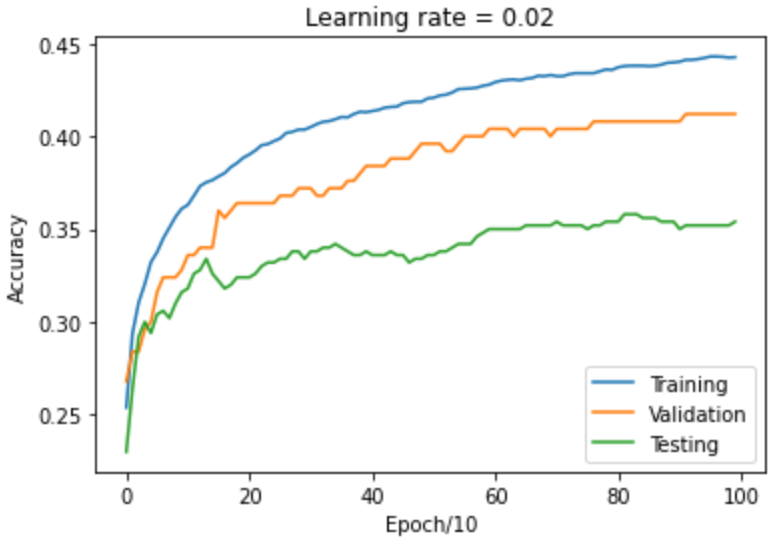

Learning rate = 0.01
Accuracy
Epoch/10
Training
Validation
Testing

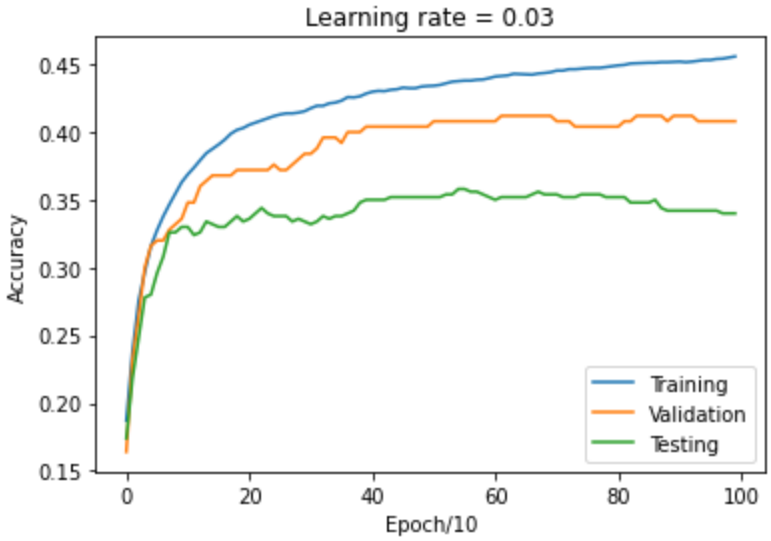

Learning rate = 0.05
Accuracy
Epoch/10
Training
Validation
Testing

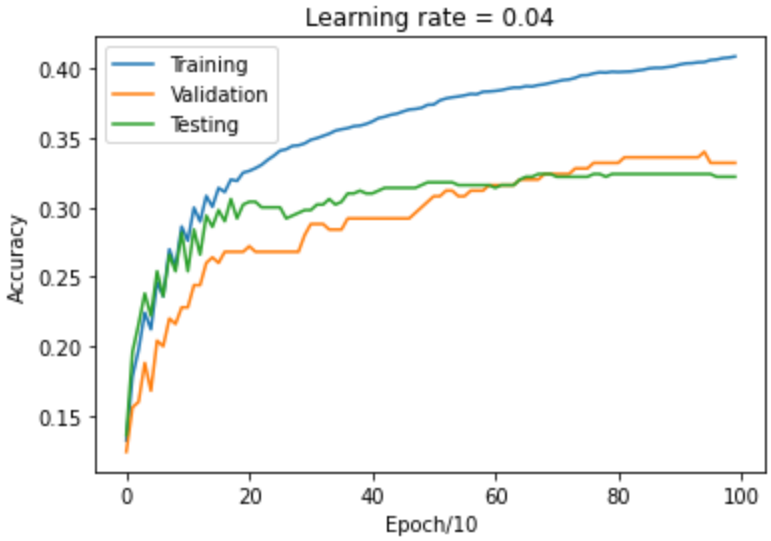


In [17]:

```
#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.

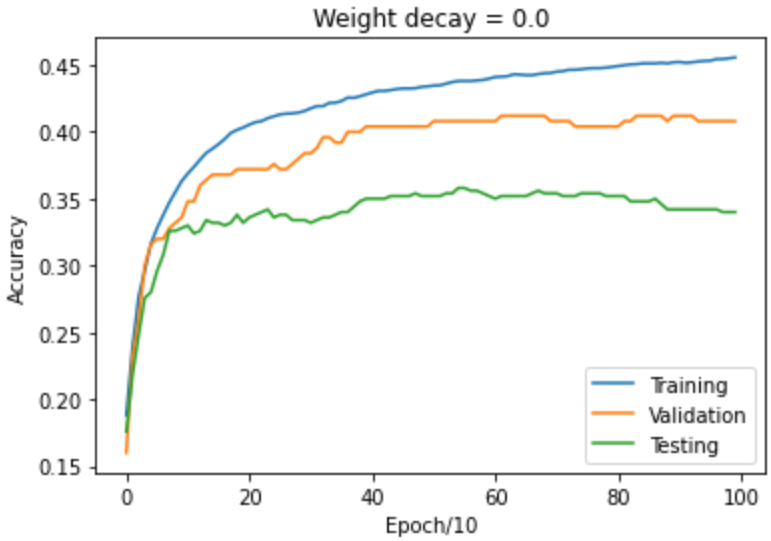
Your Answer:

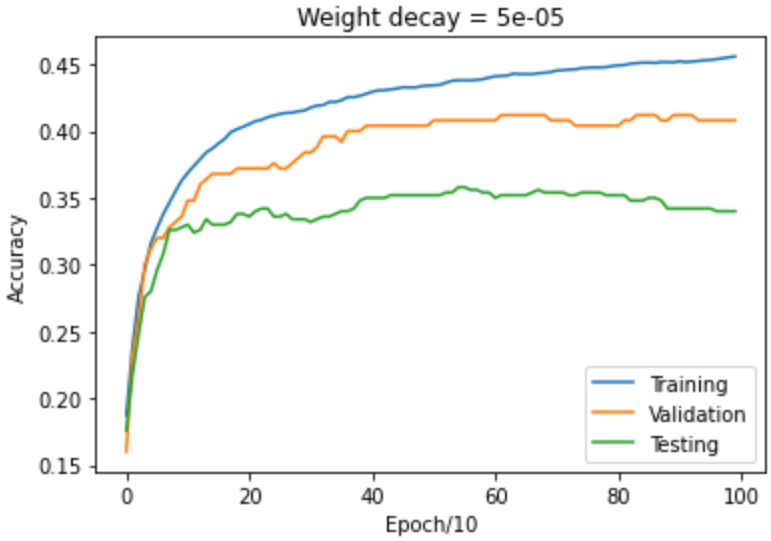
Form the graphs above, the best learning rate is 0.03. First, it does not make the model diverge. Second, it has a good learning rate, which reaches highest training accuracy (around 45%) and testing accuracy(around 35%) in 1000 epoches.

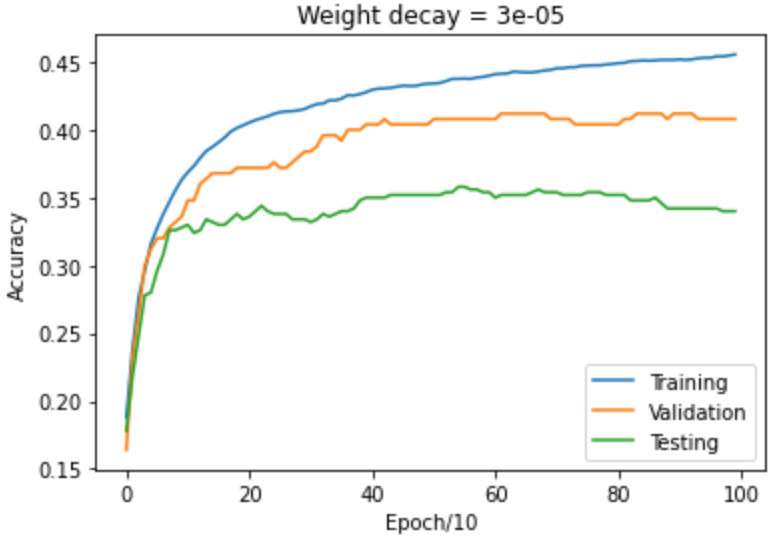
Regularization: Try different weight decay and plots graphs for all (20%)

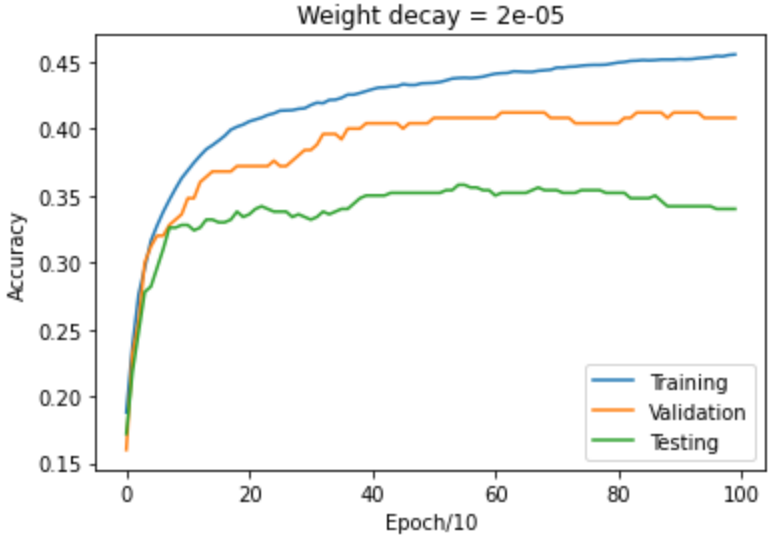
In [20]:

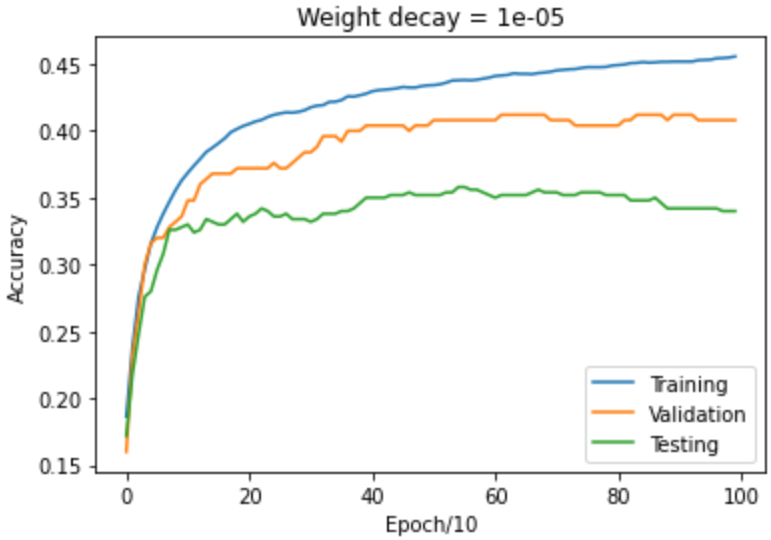
```
# Initialize a non-zero weight_decay (Regulzarization constant) term and repeat the traini
# Use the best learning rate as obtained from the above excercise, best_lr
weight_decays = [0.0, 0.00005, 0.00003, 0.00002, 0.00001, 0.000005]

# 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)
```

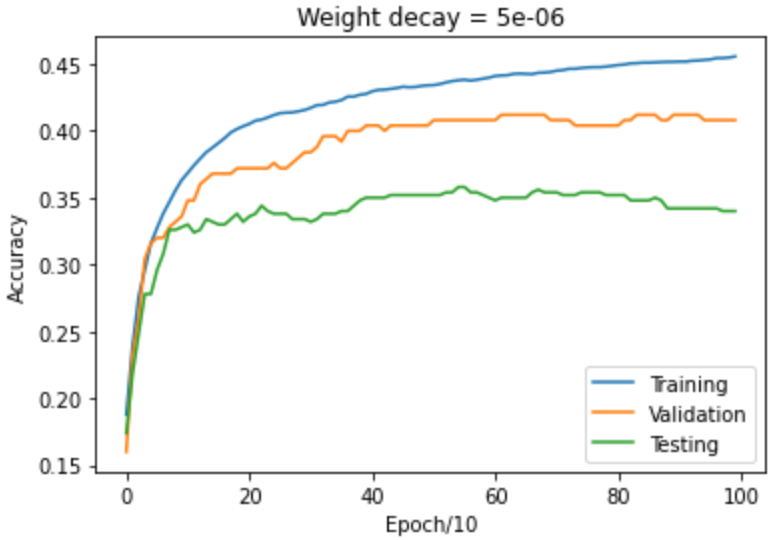



Weight decay = 5e-05
Accuracy
Epoch/10
Training
Validation
Testing


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

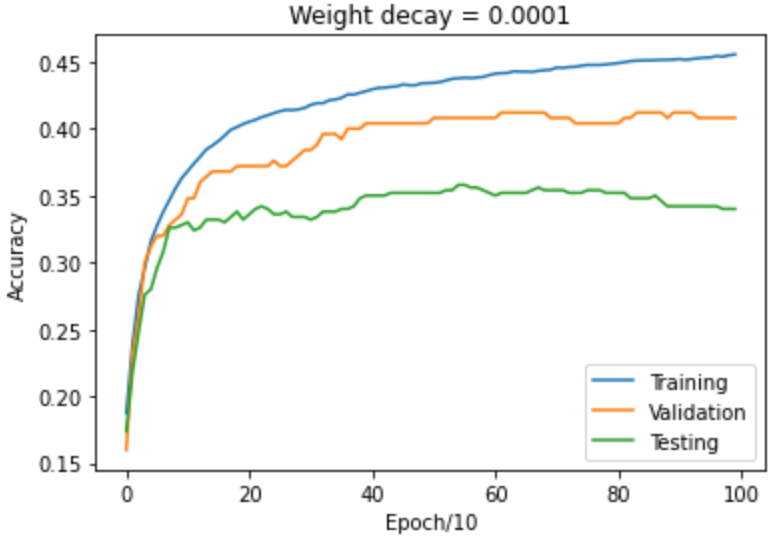

Weight decay = 2e-05
Accuracy
Epoch/10
Training
Validation
Testing

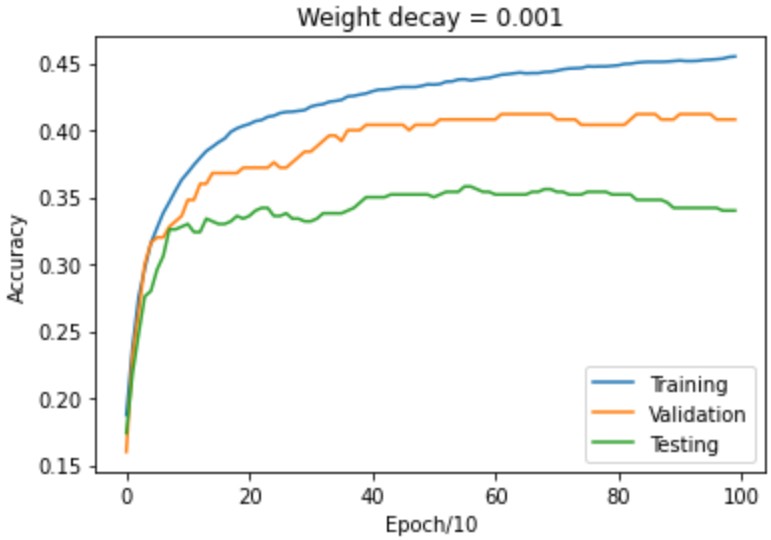


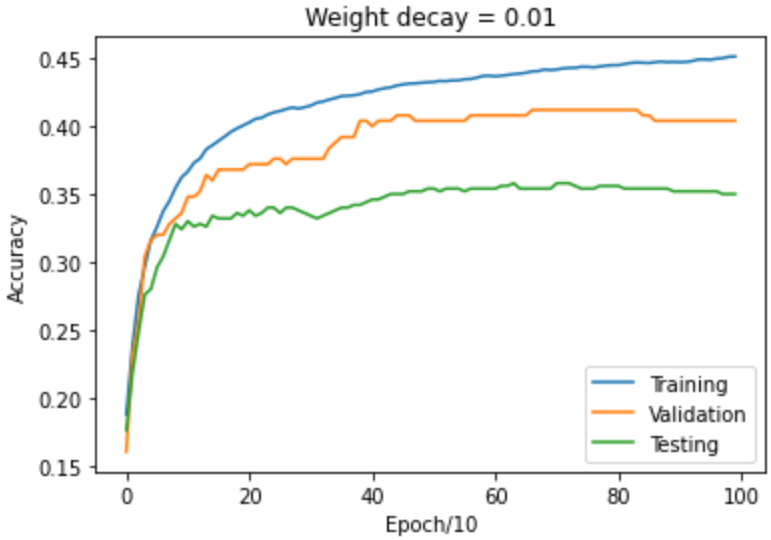


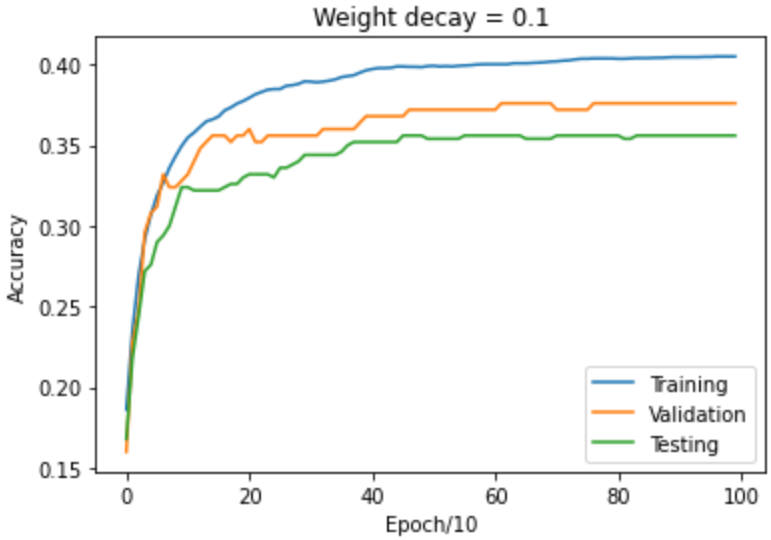
In [21]:

```
# 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:

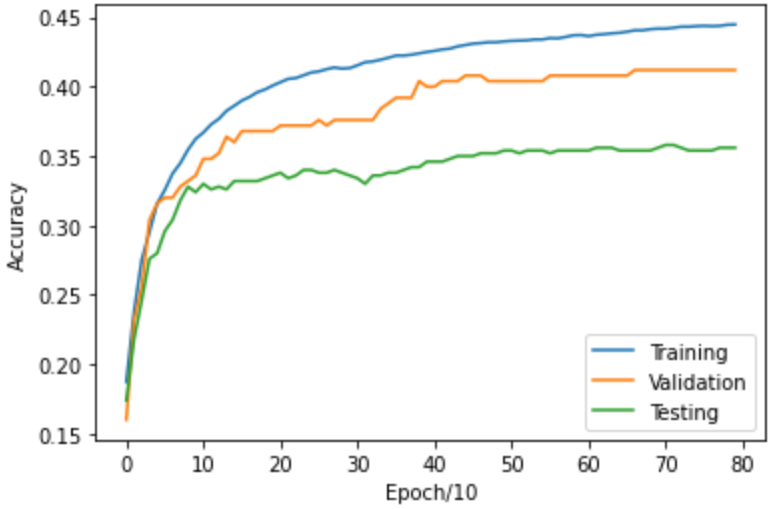
All these graphs have a higher accuracy in training set than the test set, and as the training accuracy increases, the testing accuracy decreases after a certain point. This shows that there is some overfitting. I think the best weight decay is 0.01, because it has better validation accuracy and testing 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]:

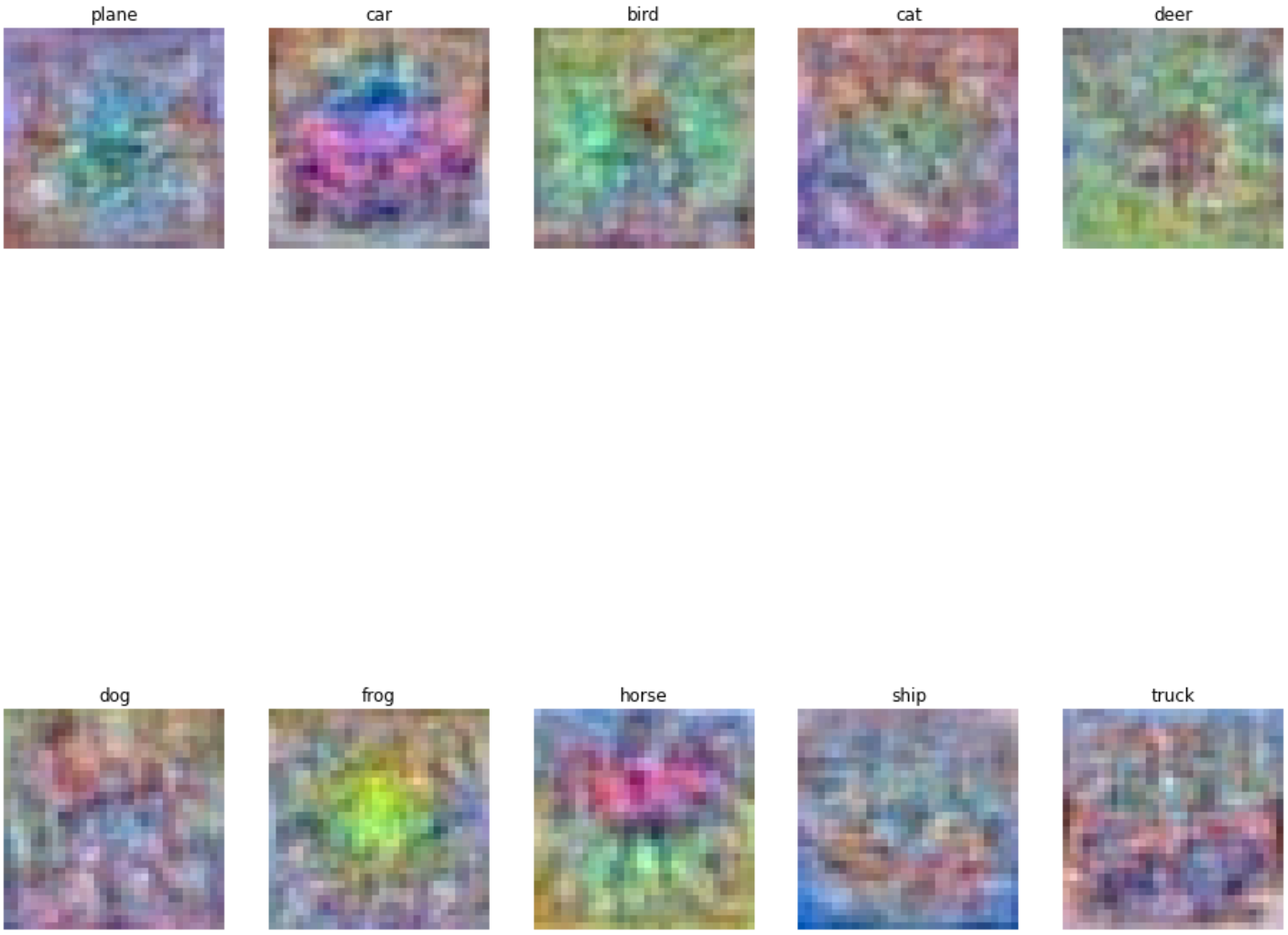
```
learning_rate = 0.03  # You will be later asked to experiment with different learning rate
num_epochs_total = 800  # Total number of epochs to train the classifier
epochs_per_evaluation = 10  # Epochs per step of evaluation; We will evaluate our model re
weight_decay = 0.01

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.

ECE285 Assignment 1: Neural Network in NumPy

Use this notebook to build your neural network by implementing the following functions in the python files under `ece285/algorithms` directory:

1. `linear.py`
2. `relu.py`
3. `softmax.py`
4. `loss_func.py`

You will be testing your 2 layer neural network implementation on a toy dataset.

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

In [85]:

```
# Setup
import matplotlib.pyplot as plt
import numpy as np

from ece285.layers.sequential import Sequential
from ece285.layers.linear import Linear
from ece285.layers.relu import ReLU
from ece285.layers.softmax import Softmax
from ece285.layers.loss_func import CrossEntropyLoss
from ece285.utils.optimizer import SGD

%matplotlib inline
plt.rcParams["figure.figsize"] = (10.0, 8.0)  # set default size of plots

# For auto-reloading external modules
# See http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

We will use the class Sequential as implemented in the file assignment2/layers/sequential.py to build a layer by layer model of our neural network. Below we initialize the toy model and the toy random data that you will use to develop your implementation.

In [86]:

```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3  # Output
num_inputs = 10  # N


def init_toy_model():
    np.random.seed(0)
    l1 = Linear(input_size, hidden_size)
    l2 = Linear(hidden_size, num_classes)

    r1 = ReLU()
    softmax = Softmax()
    return Sequential([l1, r1, l2, softmax])
```

```
def init_toy_data():
    np.random.seed(0)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.random.randint(num_classes, size=num_inputs)
    # y = np.array([0, 1, 2, 2, 1])
    return X, y


net = init_toy_model()
X, y = init_toy_data()
```

Forward Pass: Compute Scores (20%)

Implement the forward functions in Linear, Relu and Softmax layers and get the output by passing our toy data X
The output must match the given output scores

In [87]:

```
scores = net.forward(X)
print("Your scores:")
print(scores)
print()
print("correct scores:")
correct_scores = np.asarray(
    [
        [0.33333514, 0.33333826, 0.33332661],
        [0.3333351, 0.33333828, 0.33332661],
        [0.3333351, 0.33333828, 0.33332662],
        [0.3333351, 0.33333828, 0.33332662],
        [0.33333509, 0.33333829, 0.33332662],
        [0.33333508, 0.33333829, 0.33332662],
        [0.33333511, 0.33333828, 0.33332661],
        [0.33333512, 0.33333827, 0.33332661],
        [0.33333508, 0.33333829, 0.33332662],
        [0.33333511, 0.33333828, 0.33332662],
    ]
)
print(correct_scores)

# The difference should be very small. We get < 1e-7
print("Difference between your scores and correct scores:")
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[0.33333514 0.33333826 0.33332661]
 [0.3333351  0.33333828 0.33332661]
 [0.3333351  0.33333828 0.33332662]
 [0.3333351  0.33333828 0.33332662]
 [0.33333509 0.33333829 0.33332662]
 [0.33333508 0.33333829 0.33332662]
 [0.33333511 0.33333828 0.33332661]
 [0.33333512 0.33333827 0.33332661]
 [0.33333508 0.33333829 0.33332662]
 [0.33333511 0.33333828 0.33332662]]

correct scores:
[[0.33333514 0.33333826 0.33332661]
 [0.3333351  0.33333828 0.33332661]
 [0.3333351  0.33333828 0.33332662]
 [0.3333351  0.33333828 0.33332662]
 [0.33333509 0.33333829 0.33332662]
 [0.33333508 0.33333829 0.33332662]
 [0.33333511 0.33333828 0.33332661]
 [0.33333512 0.33333827 0.33332661]
```

```
 [0.33333508 0.33333829 0.33332662]
 [0.33333511 0.33333828 0.33332662]]
Difference between your scores and correct scores:
8.799388540037256e-08
```

Forward Pass: Compute loss given the output scores from the previous step (10%)

Implement the forward function in the loss_func.py file, and output the loss value. The loss value must match the given loss value.

In [88]:
```
Loss = CrossEntropyLoss()
loss = Loss.forward(scores, y)
correct_loss = 1.098612723362578
print(loss)
# should be very small, we get < 1e-12
print("Difference between your loss and correct loss:")
print(np.sum(np.abs(loss - correct_loss)))
```

```
1.0986127233625775
Difference between your loss and correct loss:
4.440892098500626e-16
```

Backward Pass (40%)

Implement the rest of the functions in the given files. Specifically, implement the backward function in all the 4 files as mentioned in the files. Note: No backward function in the softmax file, the gradient for softmax is jointly calculated with the cross entropy loss in the loss_func.backward function.

You will use the chain rule to calculate gradient individually for each layer. You can assume that this calculated gradeint then is passed to the next layers in a reversed manner due to the Sequential implementation. So all you need to worry about is implementing the gradient for the current layer and multiply it will the incoming gradient (passed to the backward function as dout) to calculate the total gradient for the parameters of that layer.

In [89]:
```
# No need to edit anything in this block ( 20% of the above 40% )
net.backward(Loss.backward())

gradients = []
for module in net._modules:
    for para, grad in zip(module.parameters, module.grads):
        assert grad is not None, "No Gradient"
        # Print gradients of the linear layer
        print(grad.shape)
        gradients.append(grad)

# Check shapes of your gradient. Note that only the linear layer has parameters
# (4, 10) -> Layer 1 W
# (10,)   -> Layer 1 b
# (10, 3) -> Layer 2 W
# (3,)    -> Layer 2 b
```

```
(4, 10)
(10,)
(10, 3)
(3,)
```

In [90]:
```
# No need to edit anything in this block ( 20% of the above 40% )
# Now we check the values for these gradients. Here are the values for these gradients, Be
# difference, you must get difference < 1e-10
```

```
grad_w1 = np.array(
    [
        [
            -6.24320917e-05,
            3.41037180e-06,
            -1.69125969e-05,
            2.41514079e-05,
            3.88697976e-06,
            7.63842314e-05,
            -8.88925758e-05,
            3.34909890e-05,
            -1.42758303e-05,
            -4.74748560e-06,
        ],
        [
            -7.16182867e-05,
            4.63270039e-06,
            -2.20344270e-05,
            -2.72027034e-06,
            6.52903437e-07,
            8.97294847e-05,
            -1.05981609e-04,
            4.15825391e-05,
            -2.12210745e-05,
            3.06061658e-05,
        ],
        [
            -1.69074923e-05,
            -8.83185056e-06,
            3.10730840e-05,
            1.23010428e-05,
            5.25830316e-05,
            -7.82980115e-06,
            3.02117990e-05,
            -3.37645284e-05,
            6.17276346e-05,
            -1.10735656e-05,
        ],
        [
            -4.35902272e-05,
            3.71512704e-06,
            -1.66837877e-05,
            2.54069557e-06,
            -4.33258099e-06,
            5.72310022e-05,
            -6.94881762e-05,
            2.92408329e-05,
            -1.89369767e-05,
            2.01692516e-05,
        ],
    ]
)
grad_b1 = np.array(
    [
        -2.27150209e-06,
        5.14674340e-07,
        -2.04284403e-06,
        6.08849787e-07,
        -1.92177796e-06,
        3.92085824e-06,
        -5.40772636e-06,
        2.93354593e-06,
        -3.14568138e-06,
        5.27501592e-11,
    ]
)
```

```
grad_w2 = np.array(
    [
        [1.28932983e-04, 1.19946731e-04, -2.48879714e-04],
        [1.08784150e-04, 1.55140199e-04, -2.63924349e-04],
        [6.96017544e-05, 1.42748410e-04, -2.12350164e-04],
        [9.92512487e-05, 1.73257611e-04, -2.72508860e-04],
        [2.05484895e-05, 4.96161144e-05, -7.01646039e-05],
        [8.20539510e-05, 9.37063861e-05, -1.75760337e-04],
        [2.45831715e-05, 8.74369112e-05, -1.12020083e-04],
        [1.34073379e-04, 1.86253064e-04, -3.20326443e-04],
        [8.86473128e-05, 2.35554414e-04, -3.24201726e-04],
        [3.57433149e-05, 1.91164061e-04, -2.26907376e-04],
    ]
)

grad_b2 = np.array([-0.1666649, 0.13333828, 0.03332662])

difference = (
    np.sum(np.abs(gradients[0] - grad_w1))
    + np.sum(np.abs(gradients[1] - grad_b1))
    + np.sum(np.abs(gradients[2] - grad_w2))
)
+np.sum(np.abs(gradients[3] - grad_b2))
print("Difference in Gradient values", difference)
```

```
Difference in Gradient values 5.665200831813677e-12
```

Train the complete network on the toy data. (30%)

To train the network we will use stochastic gradient descent (SGD), we have implemented the optimizer for you. You do not implement any more functions in the python files. Below we implement the training procedure, you should get yourself familiar with the training process. Specifically looking at which functions to call and when.

Once you have implemented the method and tested various parts in the above blocks, run the code below to train a two-layer network on toy data. You should see your training loss decrease below 0.01.

In [ ]:

```
# Training Procedure
# Initialize the optimizer. DO NOT change any of the hyper-parameters here or above.
# We have implemented the SGD optimizer class for you here, which visits each layer sequer
# get the gradients and optimize the respective parameters.
# You should work with the given parameters and only edit your implementation in the .py i

epochs = 1000
optim = SGD(net, lr=0.1, weight_decay=0.00001)

epoch_loss = []
for epoch in range(epochs):
    # Get output scores from the network
    output_x = net(X)
    # Calculate the loss for these output scores, given the true labels
    loss = Loss.forward(output_x, y)
    # Initialize your gradients to None in each epoch
    optim.zero_grad()
    # Make a backward pass to update the internal gradients in the layers
    net.backward(Loss.backward())
    # call the step function in the optimizer to update the values of the params with the
    optim.step()
    # Append the loss at each iteration
    epoch_loss.append(loss)

    print("Epoch Loss: {:3f}".format(epoch_loss[-1]))
```

In [92]:
```
# Test your predictions. The predictions must match the labels
print(net.predict(X))
print(y)
```

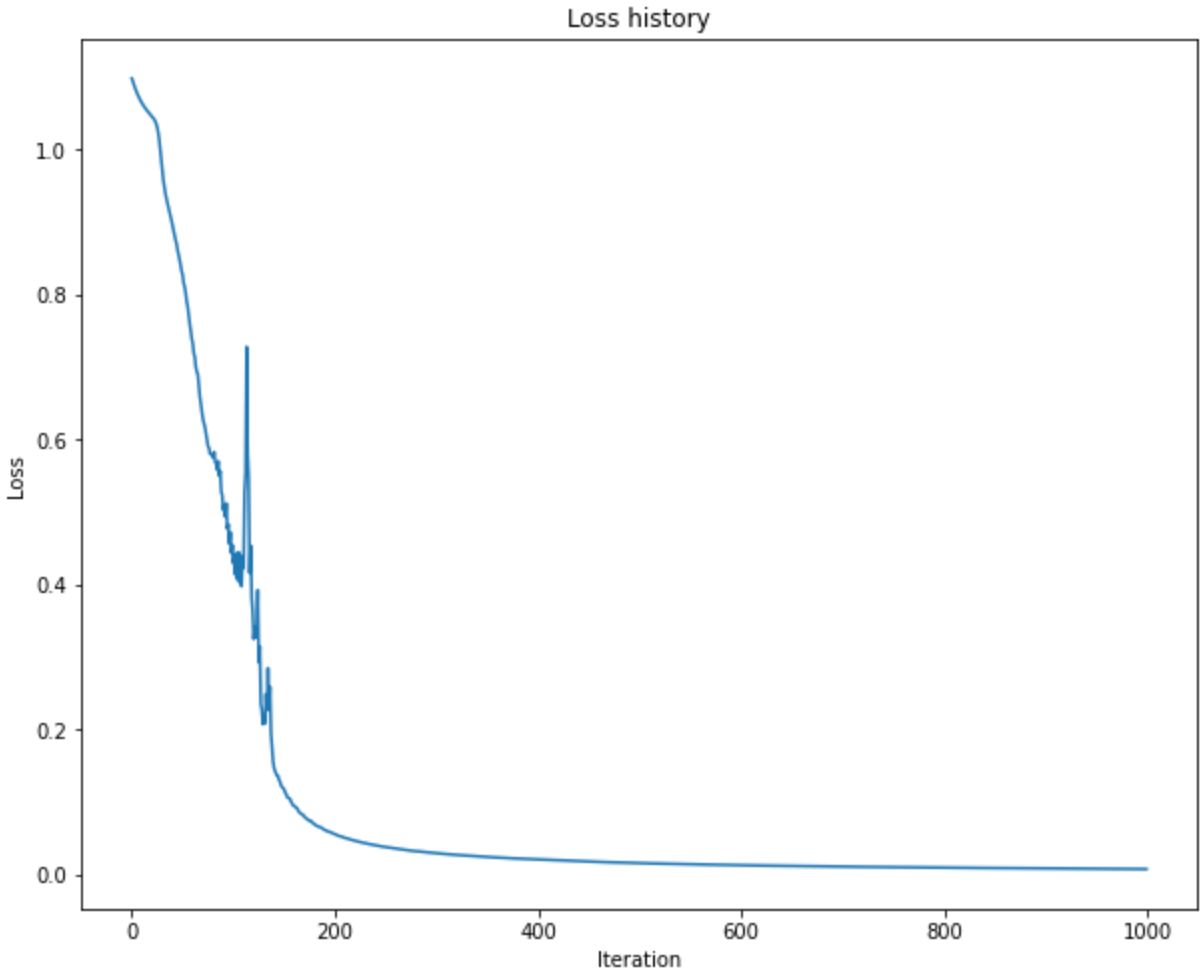
```
[2 1 0 1 2 0 0 2 0 0]
[2 1 0 1 2 0 0 2 0 0]
```

In [93]:
```
# You should be able to achieve a training loss of less than 0.02 (10%)
print("Final training loss", epoch_loss[-1])
```

```
Final training loss 0.007593419801731252
```

In [94]:
```
# Plot the training loss curve. The loss in the curve should be decreasing (20%)
plt.plot(epoch_loss)
plt.title("Loss history")
plt.xlabel("Iteration")
plt.ylabel("Loss")
```

Out[94]:
```
Text(0, 0.5, 'Loss')
```



ECE 285 Assignment 1: Classification using Neural Network

Now that you have developed and tested your model on the toy dataset set. It's time to get down and get dirty with a standard dataset such as cifar10. At this point, you will be using the provided training data to tune the hyper-parameters of your network such that it works with cifar10 for the task of multi-class classification.

Important: Recall that now we have non-linear decision boundaries, thus we do not need to do one vs all classification. We learn a single non-linear decision boundary instead. Our non-linear boundaries (thanks to relu non-linearity) will take care of differentiating between all the classes

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

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

from ece285.utils.data_processing import get_cifar10_data
from ece285.utils.evaluation import get_classification_accuracy


%matplotlib inline
plt.rcParams["figure.figsize"] = (10.0, 8.0)  # set default size of plots

# For auto-reloading external modules
# See http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

# 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)
```

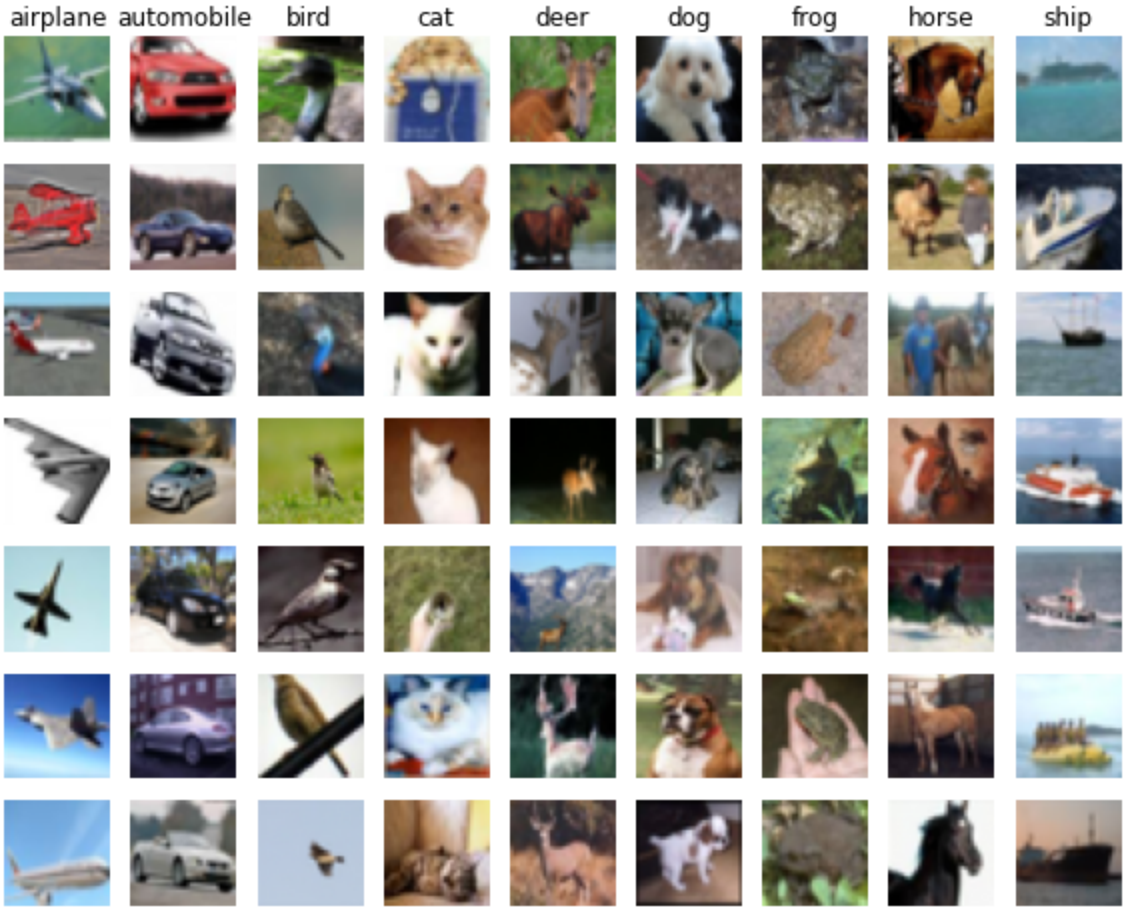
```
dict_keys(['x_train', 'y_train', 'x_val', 'y_val', 'x_test', 'y_test'])
Training Set Data  Shape:  (5000, 3072)
Training Set Label Shape:  (5000,)
Validation Set Data  Shape:  (250, 3072)
Validation Set Label Shape:  (250,)
Test Set Data  Shape:  (500, 3072)
Test Set Label Shape:  (500,)
```

In [2]:
```
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"]
```

In [6]:
```
# Import more utilies and the layers you have implemented
from ece285.layers.sequential import Sequential
from ece285.layers.linear import Linear
from ece285.layers.relu import ReLU
from ece285.layers.softmax import Softmax
from ece285.layers.loss_func import CrossEntropyLoss
from ece285.utils.optimizer import SGD
from ece285.utils.dataset import DataLoader
from ece285.utils.trainer import Trainer
```

Visualize some examples from the dataset.

In [4]:
```
# We show a few examples of training images from each class.
classes = [
    "airplane",
    "automobile",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
]
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 the first 10 classes
visualize_data(
    x_train.reshape(5000, 3, 32, 32).transpose(0, 2, 3, 1),
    classes,
    samples_per_class,
)
```



Initialize the model

In [5]:
```
input_size = 3072
hidden_size = 100  # Hidden layer size (Hyper-parameter)
num_classes = 10  # Output

# For a default setting we use the same model we used for the toy dataset.
# This tells you the power of a 2 layered Neural Network. Recall the Universal Approximati
# A 2 layer neural network with non-linearities can approximate any function, given large
def init_model():
    # np.random.seed(0) # No need to fix the seed here
    l1 = Linear(input_size, hidden_size)
    l2 = Linear(hidden_size, num_classes)

    r1 = ReLU()
    softmax = Softmax()
    return Sequential([l1, r1, l2, softmax])
```

In [7]:
```
# Initialize the dataset with the dataloader class
dataset = DataLoader(x_train, y_train, x_val, y_val, x_test, y_test)
net = init_model()
optim = SGD(net, lr=0.01, weight_decay=0.01)
loss_func = CrossEntropyLoss()
epoch = 200  # (Hyper-parameter)
batch_size = 200  # (Reduce the batch size if your computer is unable to handle it)
```

In [8]:
```
# Initialize the trainer class by passing the above modules
trainer = Trainer(
    dataset, optim, net, loss_func, epoch, batch_size, validate_interval=3
)
```

Here I deleted the print line to decrease the output space

In [9]:
```
# Call the trainer function we have already implemented for you. This trains the model fo
# hyper-parameters. It follows the same procedure as in the last ipython notebook you use
train_error, validation_accuracy = trainer.train()
```

Print the training and validation accuracies for the default hyper-parameters provided

In [10]:
```
from ece285.utils.evaluation import get_classification_accuracy

out_train = net.predict(x_train)
acc = get_classification_accuracy(out_train, y_train)
print("Training acc: ", acc)
out_val = net.predict(x_val)
acc = get_classification_accuracy(out_val, y_val)
print("Validation acc: ", acc)
```

```
Training acc:  0.3532
Validation acc:  0.324
```

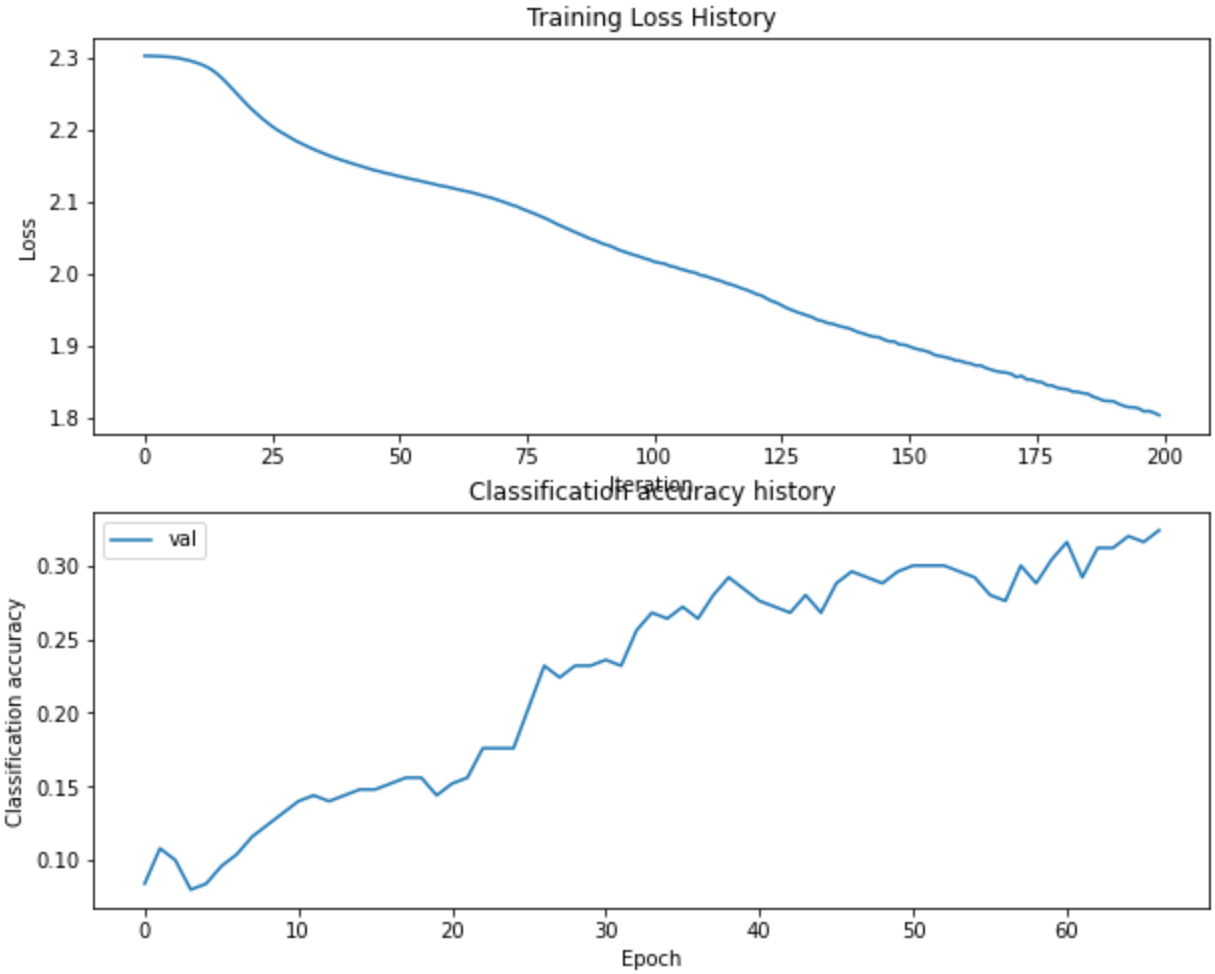
Debug the training

With the default parameters we provided above, you should get a validation accuracy of around ~0.2 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the training loss function and the validation accuracies during optimization.

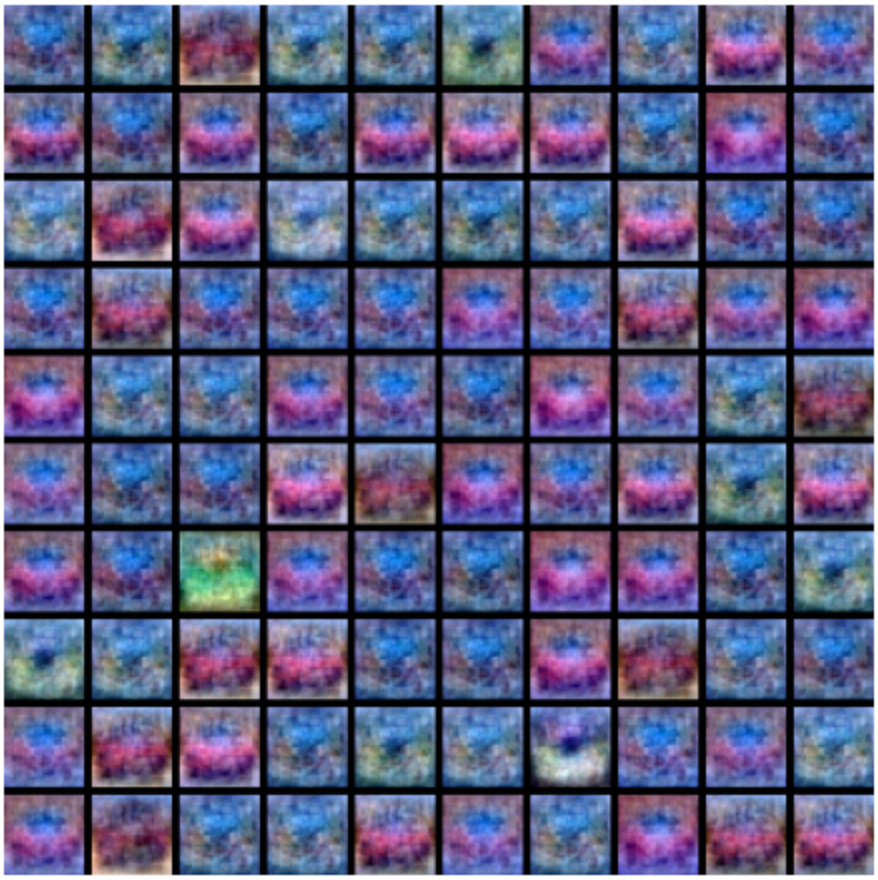
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [11]:
```
# Plot the training loss function and validation accuracies
plt.subplot(2, 1, 1)
plt.plot(train_error)
plt.title("Training Loss History")
plt.xlabel("Iteration")
plt.ylabel("Loss")

plt.subplot(2, 1, 2)
# plt.plot(stats['train_acc_history'], label='train')
plt.plot(validation_accuracy, label="val")
plt.title("Classification accuracy history")
plt.xlabel("Epoch")
plt.ylabel("Classification accuracy")
plt.legend()
plt.show()
```



In [12]:

```
from ece285.utils.vis_utils import visualize_grid

# Credits: http://cs231n.stanford.edu/

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net._modules[0].parameters[0]
    W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
    plt.imshow(visualize_grid(W1, padding=3).astype("uint8"))
    plt.gca().axis("off")
    plt.show()


show_net_weights(net)
```



Tune your hyperparameters (50%)

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength.

Approximate results. You should be aim to achieve a classification accuracy of greater than 40% on the validation set. Our best network gets over 40% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on cifar10 as you can (40% could serve as a reference), with a fully-connected Neural Network.

Explain your hyperparameter tuning process below.

Your Answer:

In [ ]:

```
input_size = 3072
hidden_sizes = [150,200,250,300,350]  # Hidden layer size (Hyper-parameter)
num_classes = 10  # Output
epoches = [500]
lrs = [0.01,0.025,0.05,0.1]
weights = [0.0001,0.001,0.01]
```

```
def init_model(hidden_size):
    # np.random.seed(0) # No need to fix the seed here
    l1 = Linear(input_size, hidden_size)
    l2 = Linear(hidden_size, num_classes)

    r1 = ReLU()
    softmax = Softmax()
    return Sequential([l1, r1, l2, softmax])
for i in hidden_sizes:
    for j in epoches:
        for l in lrs:
            for w in weights:
                net = init_model(hidden_size=i)
                optim = SGD(net, lr=l, weight_decay=w)
                loss_func = CrossEntropyLoss()
                epoch = j # (Hyper-parameter)
                batch_size = 200  # (Reduce the batch size if your computer is unable to h

                # Initialize the trainer class by passing the above modules
                trainer = Trainer(
                    dataset, optim, net, loss_func, epoch, batch_size, validate_interval=1
                )

                train_error, validation_accuracy = trainer.train()

                out_train = net.predict(x_train)
                acc = get_classification_accuracy(out_train, y_train)
                print('hidden size = {}, epoch = {}, learning rates = {}, weight decay = {
                print("Training acc: ", acc)
                out_val = net.predict(x_val)
                acc = get_classification_accuracy(out_val, y_val)
                print("Validation acc: ", acc)

                plt.subplot(1, 2, 1)
                plt.plot(train_error)
                plt.title("Training Loss History")
                plt.xlabel("Iteration")
                plt.ylabel("Loss")

                plt.subplot(1, 2, 2)
                # plt.plot(stats['train_acc_history'], label='train')
                plt.plot(validation_accuracy, label="val")
                plt.title("Classification accuracy history")
                plt.xlabel("Epoch")
                plt.ylabel("Classification accuracy")
                plt.legend()
                plt.show()
```

In [ ]:

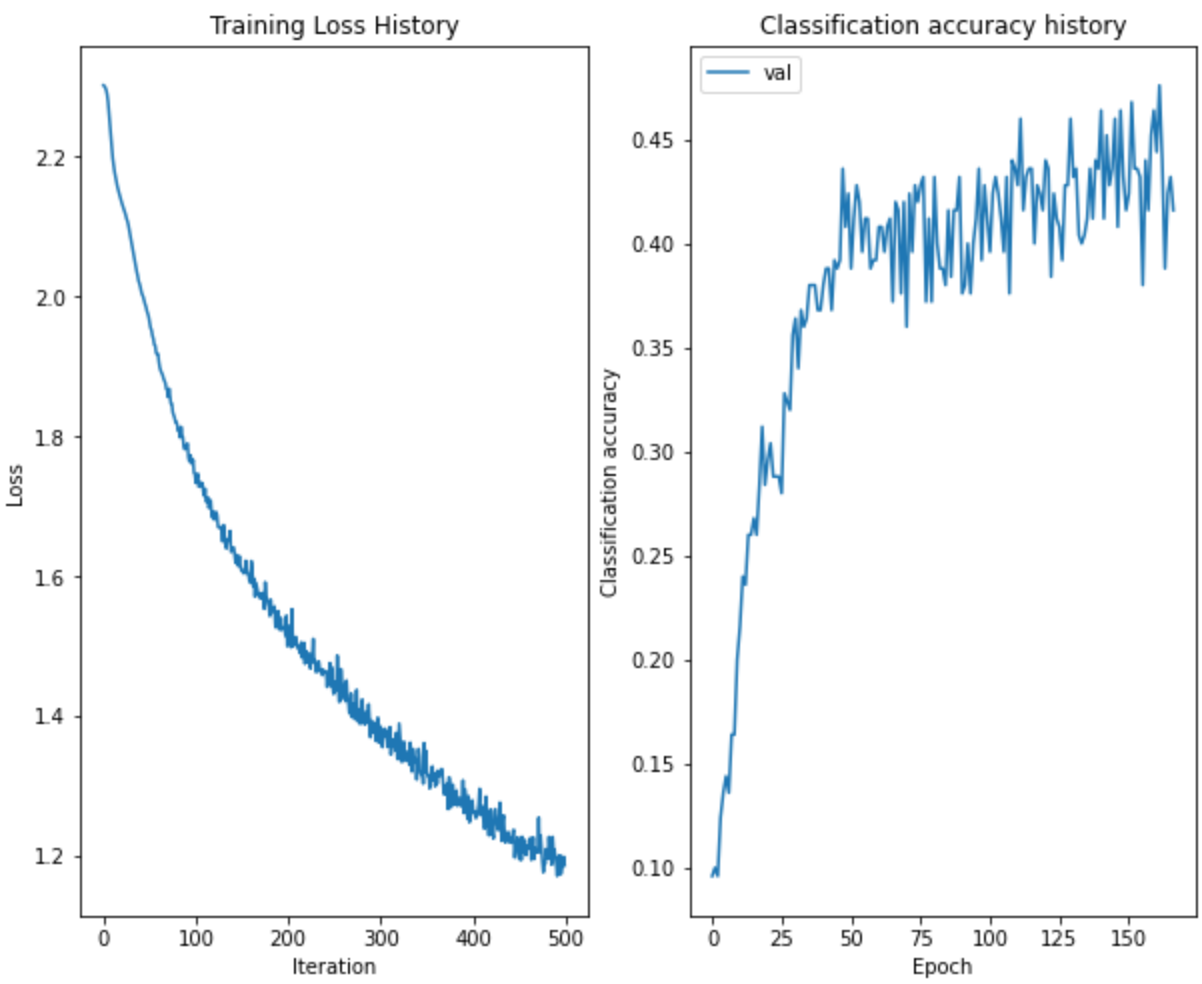
```
best_net_hyperparams = [0.025, 0.01, 500, 200]  # store the best model into this

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model hyperparams in best_net.                                                #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# You are now free to test different combinations of hyperparameters to build   #
# various models and test them according to the above plots and visualization   #


# TODO: Show the above plots and visualizations for the default params (already #
```

```
# done) and the best hyper-params you obtain. You only need to show this for 2  #
# sets of hyper-params.                                                          #
# You just need to store values for the hyperparameters in best_net_hyperparams #
# as a list in the order
# best_net_hyperparams = [lr, weight_decay, epoch, hidden_size]
################################################################################

pass
```

In [24]:

```
# TODO: Plot the training_error and validation_accuracy of the best network (5%)

best_net = init_model(hidden_size=200)
optim = SGD(best_net, lr=0.025, weight_decay=0.01)
loss_func = CrossEntropyLoss()
epoch = 500 # (Hyper-parameter)
batch_size = 200  # (Reduce the batch size if your computer is unable to handle it)

# Initialize the trainer class by passing the above modules
trainer = Trainer(
    dataset, optim, best_net, loss_func, epoch, batch_size, validate_interval=3
)

train_error, validation_accuracy = trainer.train()

out_train = best_net.predict(x_train)
acc = get_classification_accuracy(out_train, y_train)
print('hidden size = {}, epoch = {}, learning rates = {}, weight decay = {}'.format(i,j,l,
print("Training acc: ", acc)
out_val = best_net.predict(x_val)
acc = get_classification_accuracy(out_val, y_val)
print("Validation acc: ", acc)

plt.subplot(1, 2, 1)
plt.plot(train_error)
plt.title("Training Loss History")
plt.xlabel("Iteration")
plt.ylabel("Loss")

plt.subplot(1, 2, 2)
# plt.plot(stats['train_acc_history'], label='train')
plt.plot(validation_accuracy, label="val")
plt.title("Classification accuracy history")
plt.xlabel("Epoch")
plt.ylabel("Classification accuracy")
plt.legend()
plt.show()
# TODO: visualize the weights of the best network (5%)
```

```
hidden size = 200, epoch = 500, learning rates = 0.01, weight decay = 0.001
Training acc:  0.6098
Validation acc:  0.392
```



In [25]:
```
show_net_weights(best_net)
```

Run on the test set (30%)

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 35%.

In [26]:
```
test_acc = (best_net.predict(x_test) == y_test).mean()
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.42
```

Inline Question (10%)

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

Your Answer:

1. Train on a larger dataset
2. Increase the regularization strength

Your Explanation:

Testing accuracy being much lower than the training accuracy means overfitting. That's to say, the complexity of the model is too much and the data is not capable of training it. To solve this, the model shall be trained on a larger dataset, and the regularization strength shall be added.