ECE 176 Assignment 2: 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 [2]:
```
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from 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 these hyperparameters:

- **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`. The formulations follow the lecture (consider binary classification for each of the 10 classes, with labels -1 / 1 for not belonging / belonging to the class). You need to fill in the training function as well as the prediction function. 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 Regressic
from algorithms import Logistic
from 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
num_epochs_total = 200  # Total number of epochs to train the classifier
epochs_per_evaluation = 10  # Epochs per step of evaluation; We will evaluat
N, D = dataset[
    "x_train"
].shape  # Get training data shape, N: Number of examples, D:Dimensionality
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()
```

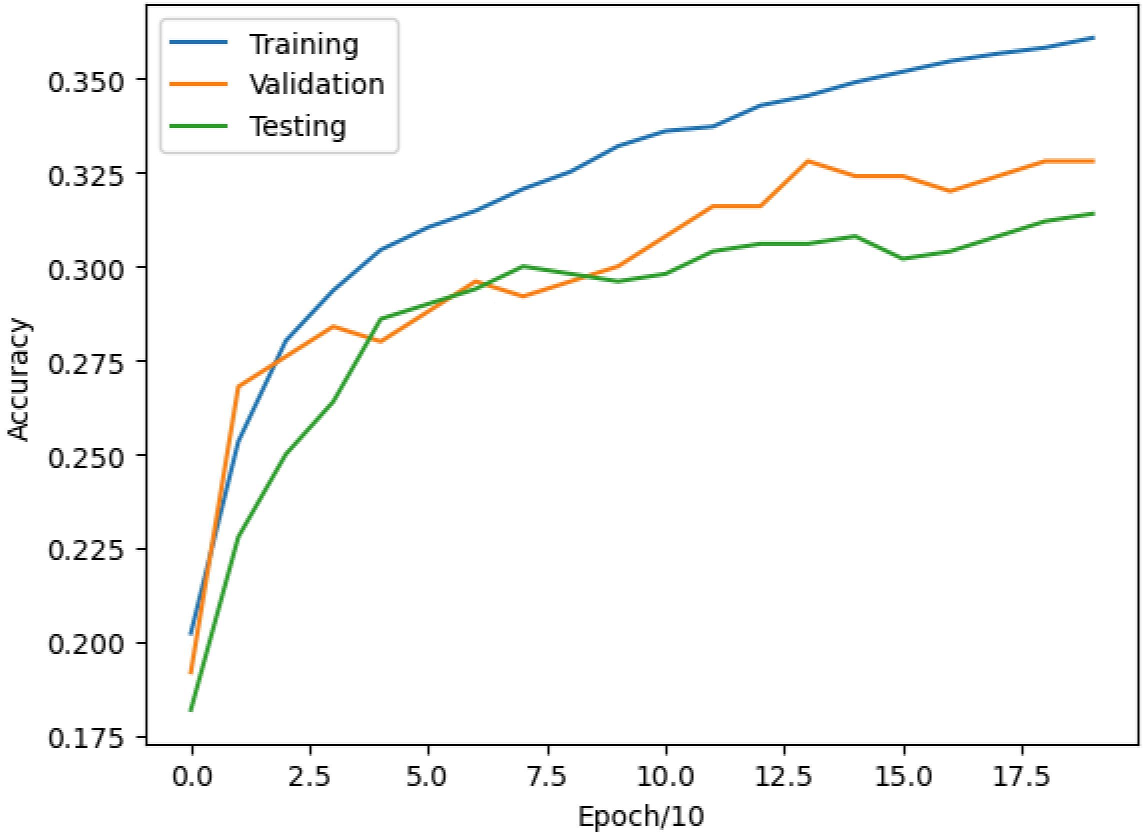
```
# Insert additional scalar term 1 in the samples to account for the bias as
x_train = np.insert(x_train, D, values=1, axis=1)
x_val = np.insert(x_val, D, values=1, axis=1)
x_test = np.insert(x_test, D, values=1, axis=1)
```

In [9]:
```
# 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_

        # 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_te

    return train_accuracies, val_accuracies, test_accuracies, weights
```

In [11]:
```
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 [13]:
```
# Run training and plotting for default parameter values as mentioned above
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
```

In [15]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```

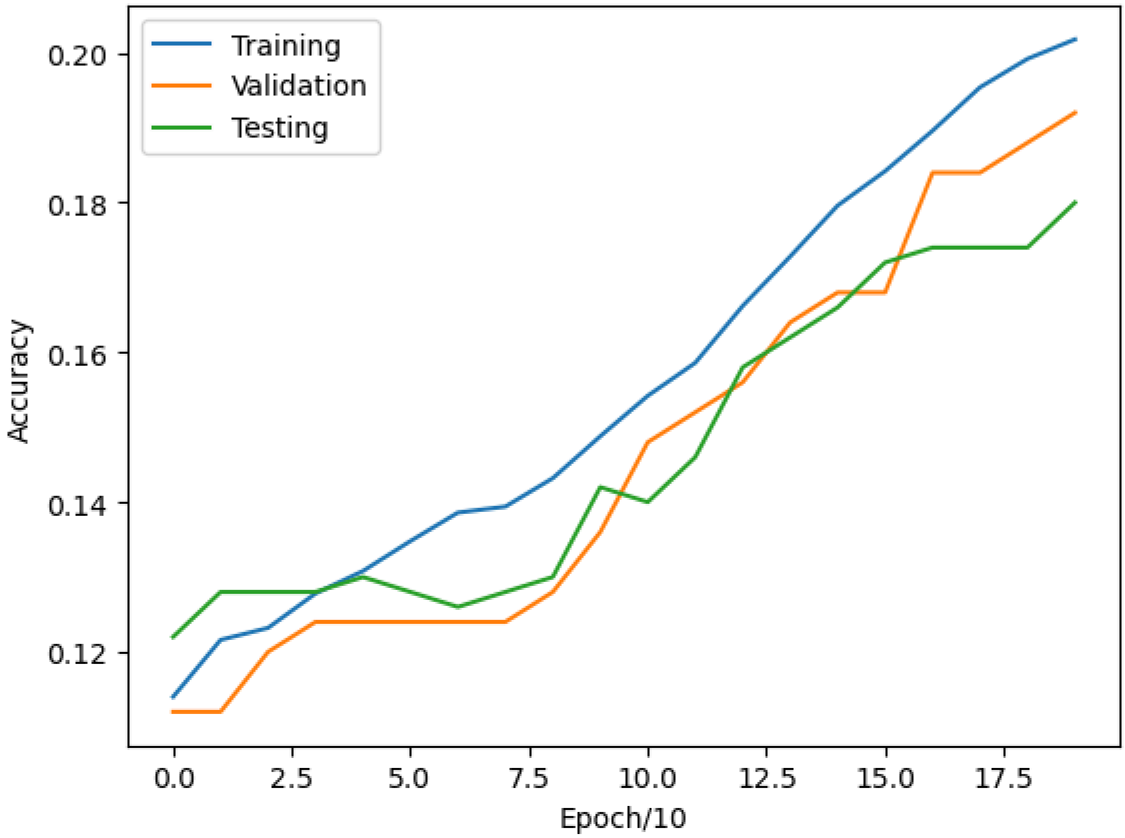


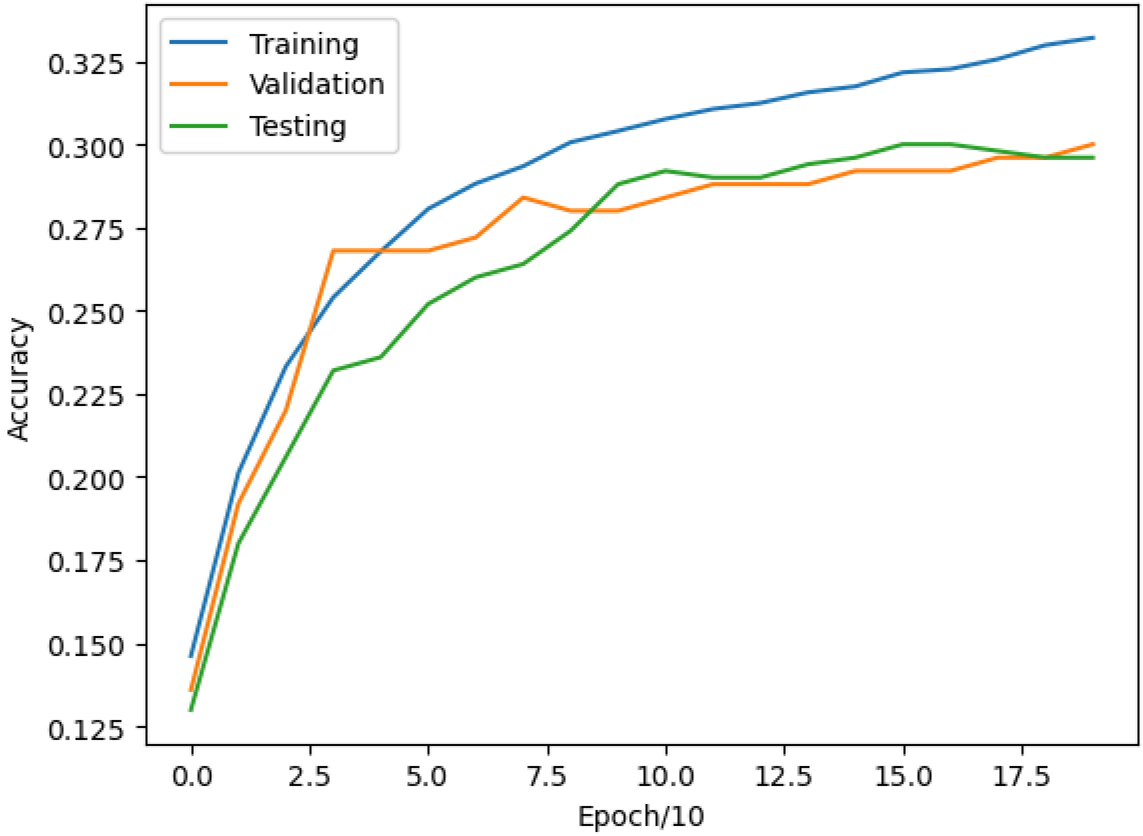
Try different learning rates and plot graphs for all (20%)

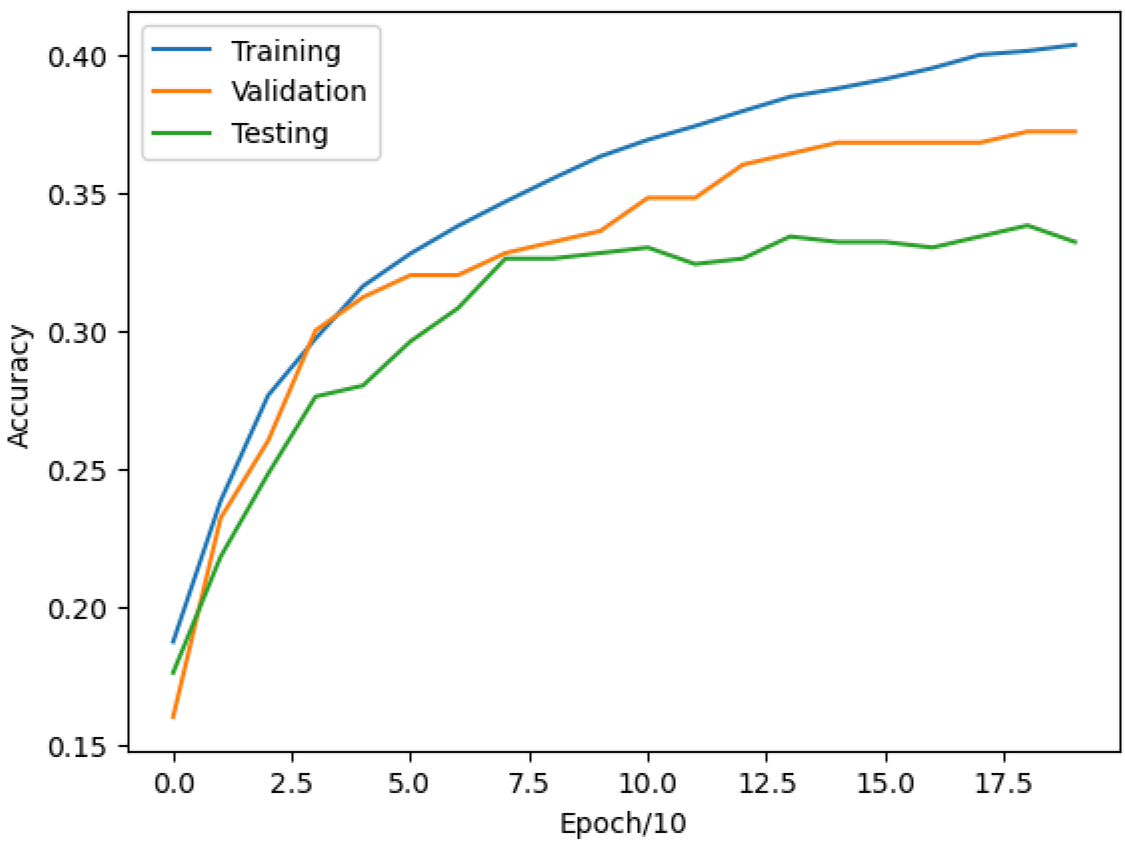
In [38]:
```
# Initialize the best values
best_weights = weights
best_learning_rate = learning_rate
best_weight_decay = weight_decay

# TODO
# Repeat the above training and evaluation steps for the following learning
# You need to try 3 learning rates and submit all 3 graphs along with this r
learning_rates = [0.0005,0.005,0.03]
weight_decay = 0.0  # No regularization for now

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY ACH

# 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)
train_accuracies=[]
val_accuracies=[]
test_accuracies=[]
for learning_rate in learning_rates:
    train_accu, val_accu, test_accu, weights = train(learning_rate, weight_c

    train_accuracies.append(train_accu)
    val_accuracies.append(val_accu)
    test_accuracies.append(test_accu)
```

```
    print(f"Learning rate:{learning_rate:.6f}")

plot_accuracies(train_accuracies[0], val_accuracies[0], test_accuracies[0])
plot_accuracies(train_accuracies[1], val_accuracies[1], test_accuracies[1])
plot_accuracies(train_accuracies[2], val_accuracies[2], test_accuracies[2])



    # TODO: Train the classifier with different learning rates and plot
    #pass
```

```
Learning rate:0.000500
Learning rate:0.005000
Learning rate:0.030000
```







Inline Question 1.

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

Your Answer:

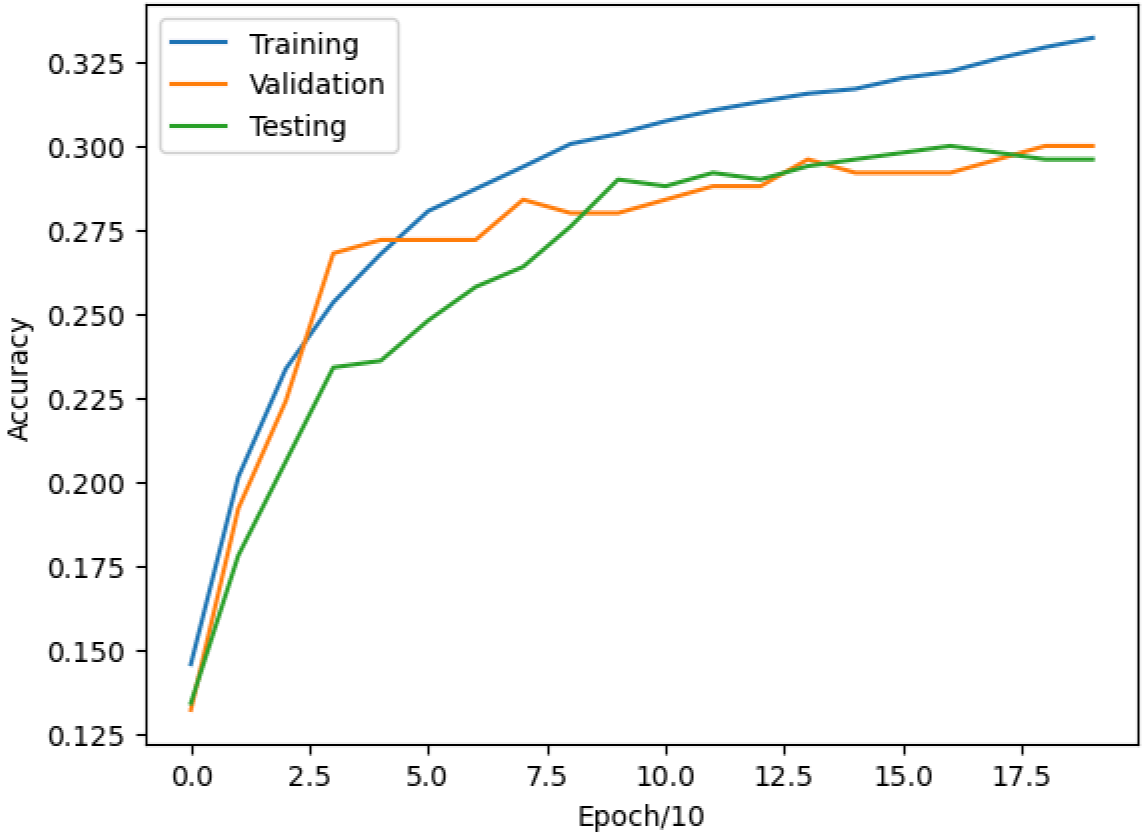
After analyzing the three learning rates (0.000500, 0.005000, and 0.030000), I would choose lr=0.005000 (middle plot) as the optimal learning rate. While the smallest learning rate (0.000500) shows slow and unstable learning with poor accuracy around 0.20, and the largest learning rate (0.030000) shows clear signs of overfitting with a large gap between training and validation curves despite reaching higher accuracy. The middle learning rate (0.005000) provides the best balance, achieving good convergence around 0.30-0.325 accuracy while maintaining small gaps between training, validation, and testing curves, indicating good generalization. This medium learning rate avoids both the underfitting issues of the slower rate and the overfitting problems of the faster rate, making it the most suitable choice for training a robust model.

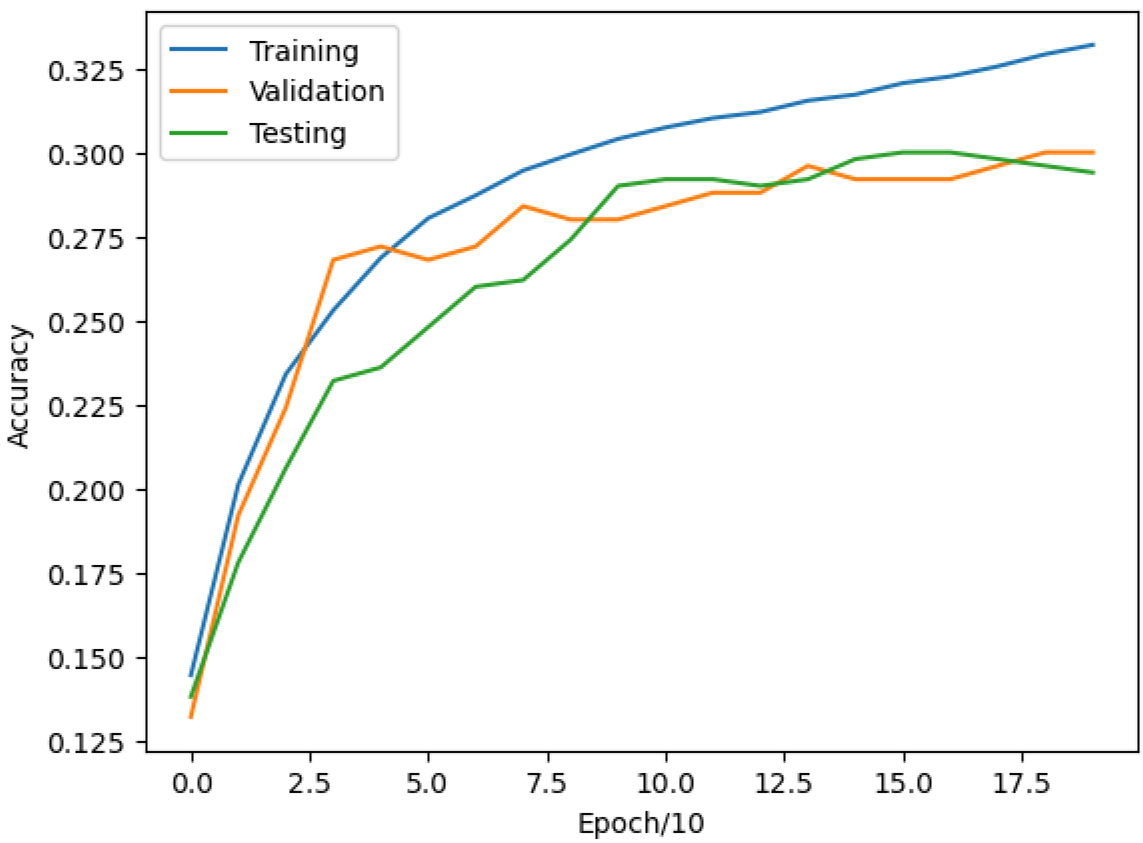
Regularization: Try different weight decay and plots graphs for all (20%)

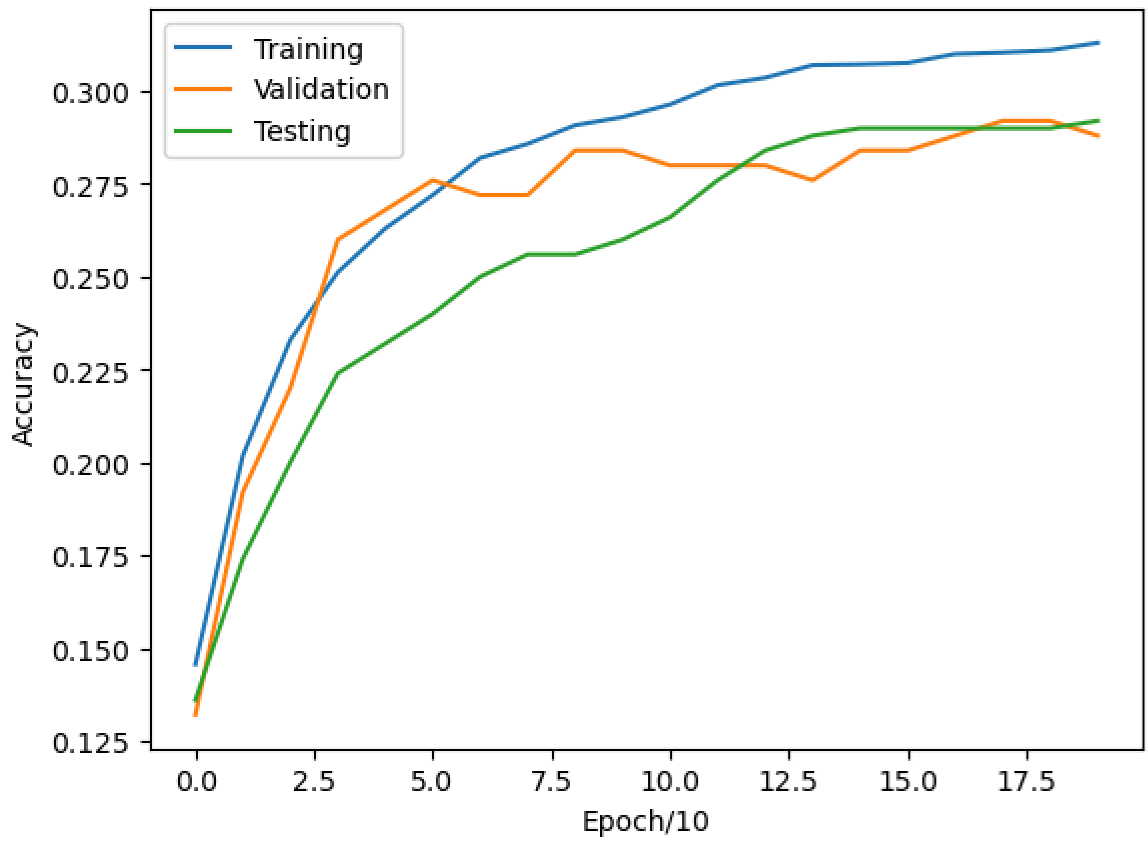
In [47]:

```
# Initialize a non-zero weight_decay (Regulzarization constant) term and rep
# Use the best learning rate as obtained from the above excercise, best_lr

# You need to try 3 learning rates and submit all 3 graphs along with this r
weight_decays = [0.00001,0.001,1]
learning_rate=0.005

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY ACH

# 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)
train_accuracies=[]
val_accuracies=[]
test_accuracies=[]
for weight_decay in weight_decays:
    train_accu, val_accu, test_accu, weights = train(learning_rate, weight_d
    train_accuracies.append(train_accu)
    val_accuracies.append(val_accu)
    test_accuracies.append(test_accu)

    print(f"weigh_decay:{weight_decay:.6f}")
plot_accuracies(train_accuracies[0], val_accuracies[0], test_accuracies[0])
plot_accuracies(train_accuracies[1], val_accuracies[1], test_accuracies[1])
plot_accuracies(train_accuracies[2], val_accuracies[2], test_accuracies[2])

    # TODO: Train the classifier with different weight decay and plot
    #pass
```

```
weigh_decay:0.000010
weigh_decay:0.001000
weigh_decay:1.000000
```


Training
Validation
Testing
0.325
0.300
0.275
0.250
0.225
0.200
0.175
0.150
0.125
Accuracy
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epoch/10


Training
Validation
Testing
0.325
0.300
0.275
0.250
0.225
0.200
0.175
0.150
0.125
Accuracy
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epoch/10



Inline Question 2.

Discuss underfitting and overfitting as observed in the 3 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

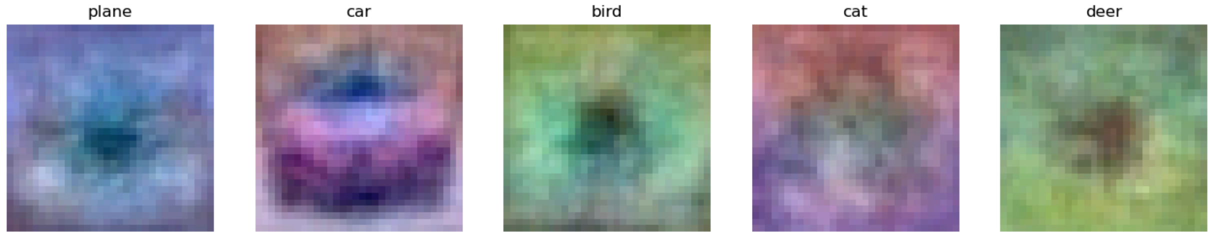
Your Answer:

After analyzing the three weight decay values (0.000010, 0.001000, 1.000000), I would choose weight_decay=0.001000 (middle plot) as the optimal regularization parameter. The smallest weight decay (0.000010) shows signs of overfitting with a growing gap between training and validation/testing curves, while the largest weight decay (1.000000) constrains the model too much, slightly reducing overall performance and potentially leading to underfitting. The middle weight decay value provides the best balance, showing good generalization with minimal gap between training and validation/test curves while maintaining strong performance around 0.30 accuracy. This middle ground in regularization ensures the model performs well on unseen data, which is the true goal of machine learning, rather than just optimizing for training performance.

Visualize the filters (10%)

In [52]:
```
# These visualizations will only somewhat make sense if your learning rate a
# properly chosen in the model. Do your best.

# TODO: Run this cell and Show filter visualizations for the best set of wei
# Report the 2 hyperparameters you used to obtain the best model.

# NOTE: You need to set `best_learning_rate` and `best_weight_decay` to the
best_learning_rate=0.005
best_weight_decay=0.001
print("Best LR:", best_learning_rate)
print("Best Weight Decay:", best_weight_decay)

# NOTE: You need to set `best_weights` to the weights with the highest accur
w = best_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()
```

```
Best LR: 0.005
Best Weight Decay: 0.001
```



Inline Question 3. (9%)

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?

Comparing both models, Linear Regression (first graph) shows higher accuracy around 0.35-0.38 but with unstable validation performance and larger gaps between training and testing curves, indicating potential overfitting. In contrast, Logistic Regression (second graph) demonstrates more stable learning with consistent performance across all sets, reaching accuracy around 0.30-0.325, and showing better generalization despite slightly lower overall accuracy. For a multiclass classification project, I would still choose Logistic Regression because although Linear Regression shows higher accuracy, its unstable validation curve and larger gaps between curves suggest it might not perform reliably on new, unseen data. The stable performance and better generalization of Logistic Regression make it a more reliable choice for real-world applications where consistent performance on new data is crucial.

Your Answer:

Survey (1%)

Question:

How many hours did you spend on assignment 2?

Your Answer:

15 hours