logistic_regression

April 28, 2023

1 ECE 285 Assignment 1: Logistic Regression

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

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

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

[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,)
```

2 Logistic Regression for multi-class classification

A Logistic Regression Algorithm has 3 hyperparameters that you can experiment with:

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

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

2.0.1 Implementation (40%)

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

[2]:
```
# Import the algorithm implementation (TODO: Complete the Logistic Regression␣
↪in algorithms/logistic_regression.py)
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 rates and report results
num_epochs_total = 1000  # Total number of epochs to train the classifier
epochs_per_evaluation = 10  # Epochs per step of evaluation; We will evaluate␣
↪our model regularly during training
N, D = dataset[
    "x_train"
].shape  # Get training data shape, N: Number of examples, D:Dimensionality of␣
↪the data
weight_decay = 0.00002

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␣
 ↪discussed in class
x_train = np.insert(x_train, D, values=1, axis=1)
x_val = np.insert(x_val, D, values=1, axis=1)
x_test = np.insert(x_test, D, values=1, axis=1)
```

[3]:

```
# 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))
        #print(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
```

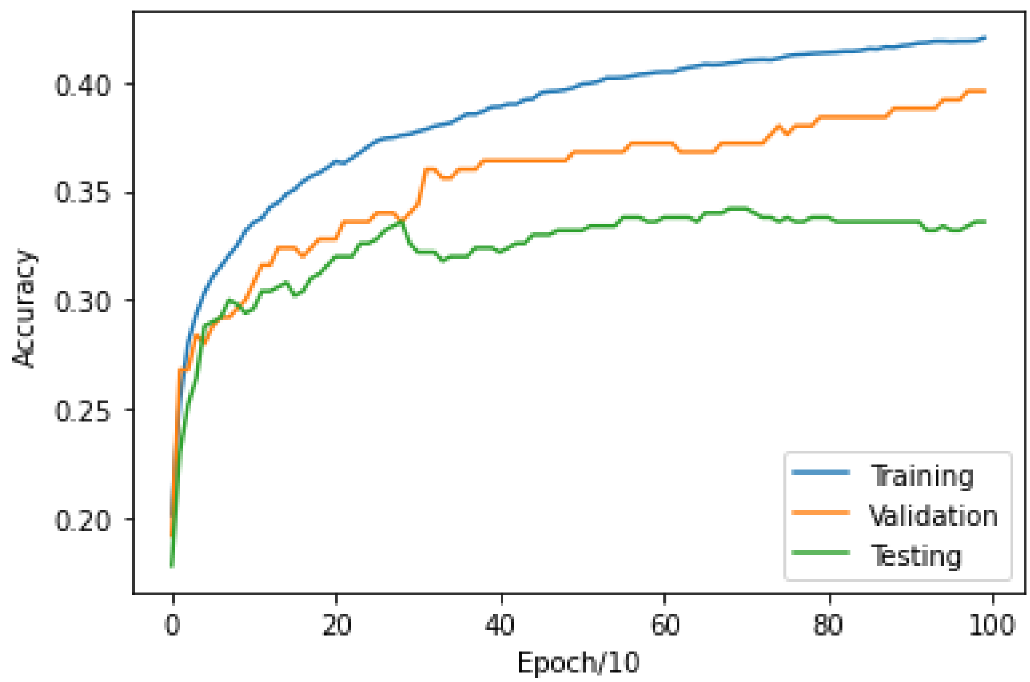
[4]:

```
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()
```

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

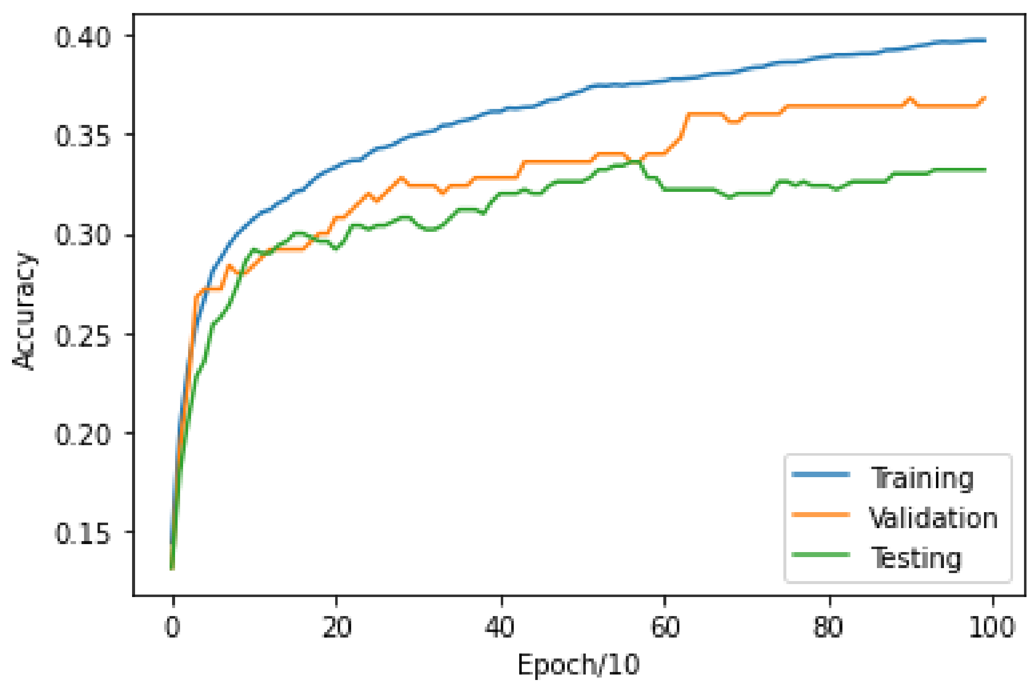
[ ]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```



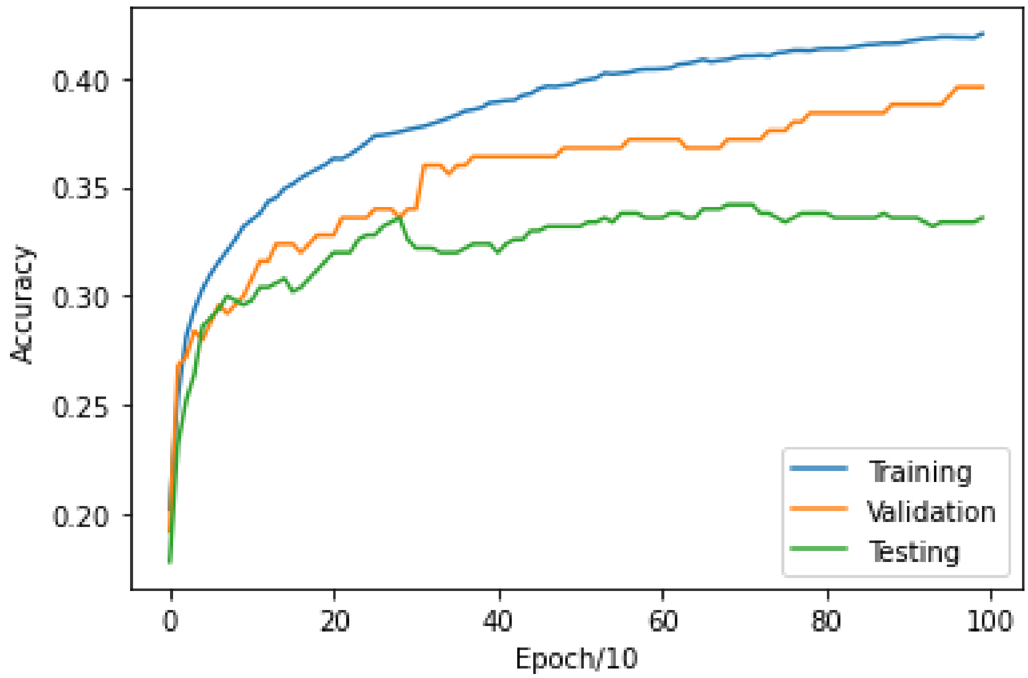
2.0.2 Try different learning rates and plot graphs for all (20%)

[7]:
```
# 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␣
↪rates and plot graphs
```

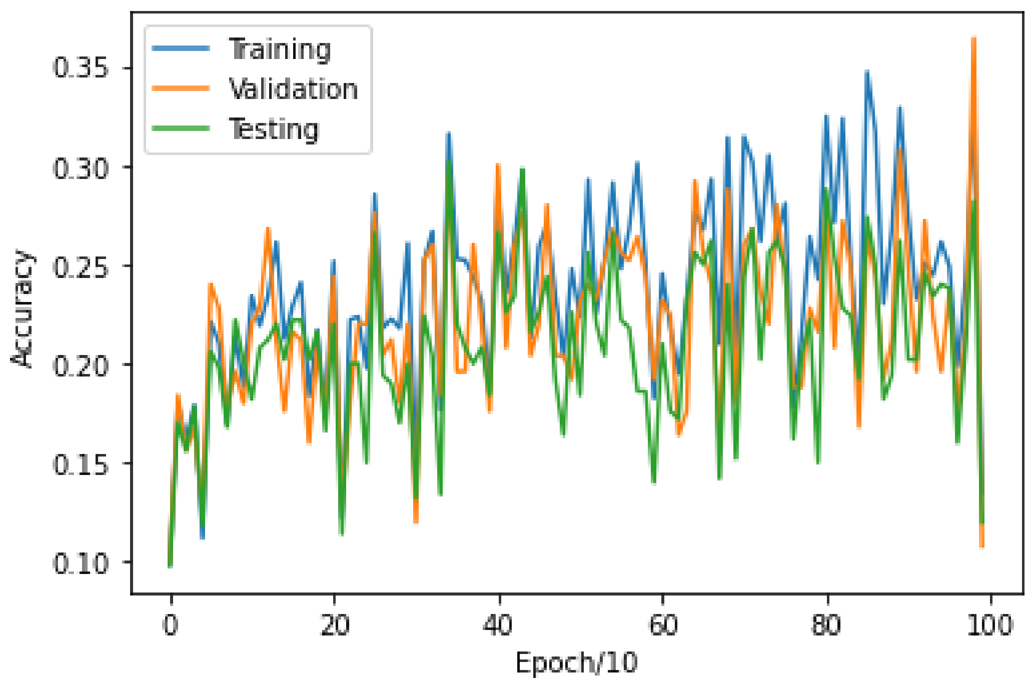
```
# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your learning rate experiments
learning_rates = [0.005, 0.01, 0.1]
weight_decay = 0.0  # No regularization for now

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

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

for learning_rate in learning_rates:
    # TODO: Train the classifier with different learning rates and plot
    t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
    plot_accuracies(t_ac, v_ac, te_ac)
    print(t_ac[-1], v_ac[-1], te_ac[-1])
```




```
0.397 0.368 0.332
```



0.4204 0.396 0.336



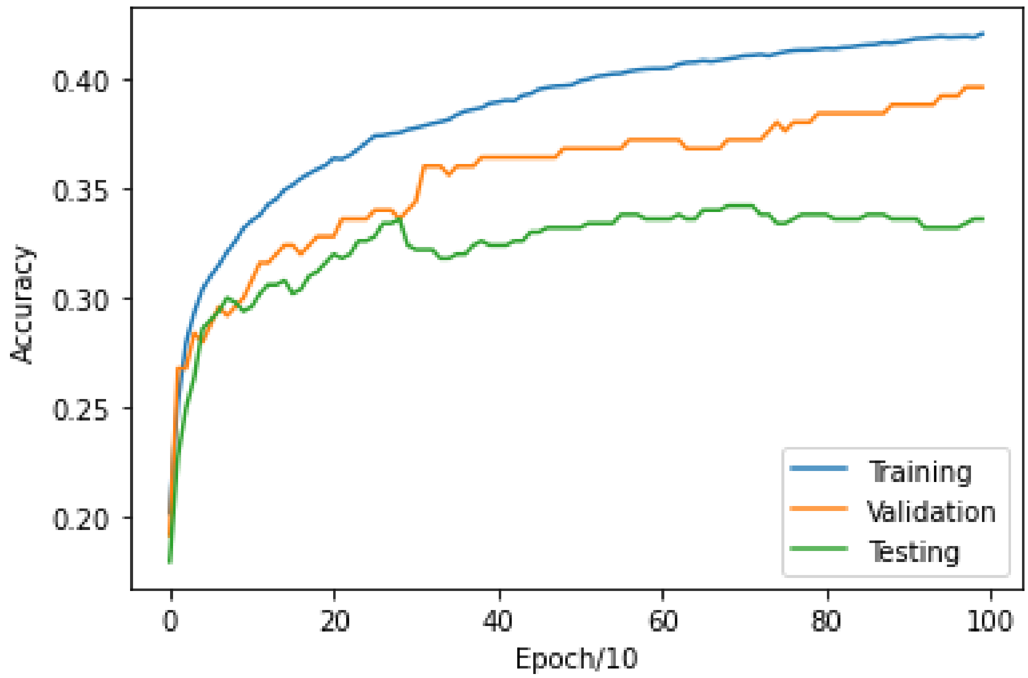
0.1344 0.108 0.12

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

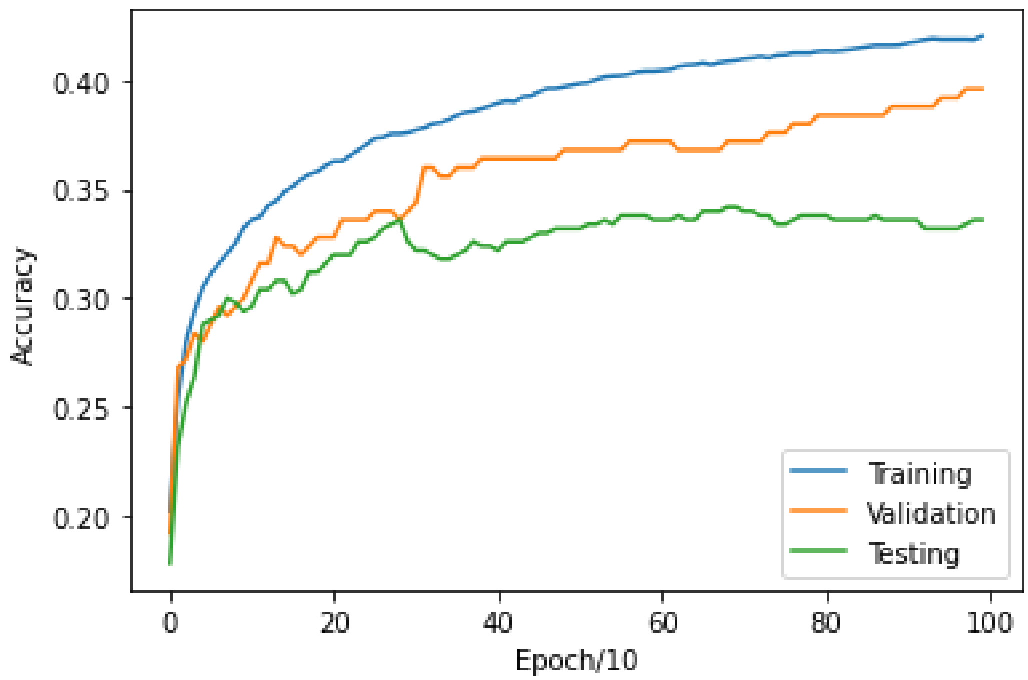
Your Answer: The best learning rate out of the above code is 0.01, and hence that would be a better pick to train the model. If the learning rate is too small, the model might converge very slowly, leading to a longer training time and less efficient optimization. Conversely, if the learning rate is too large, the algorithm might overshoot the best solution and diverge, causing the model to perform poorly. In our case out of the models tested 0.01 seems to be a better fit.

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

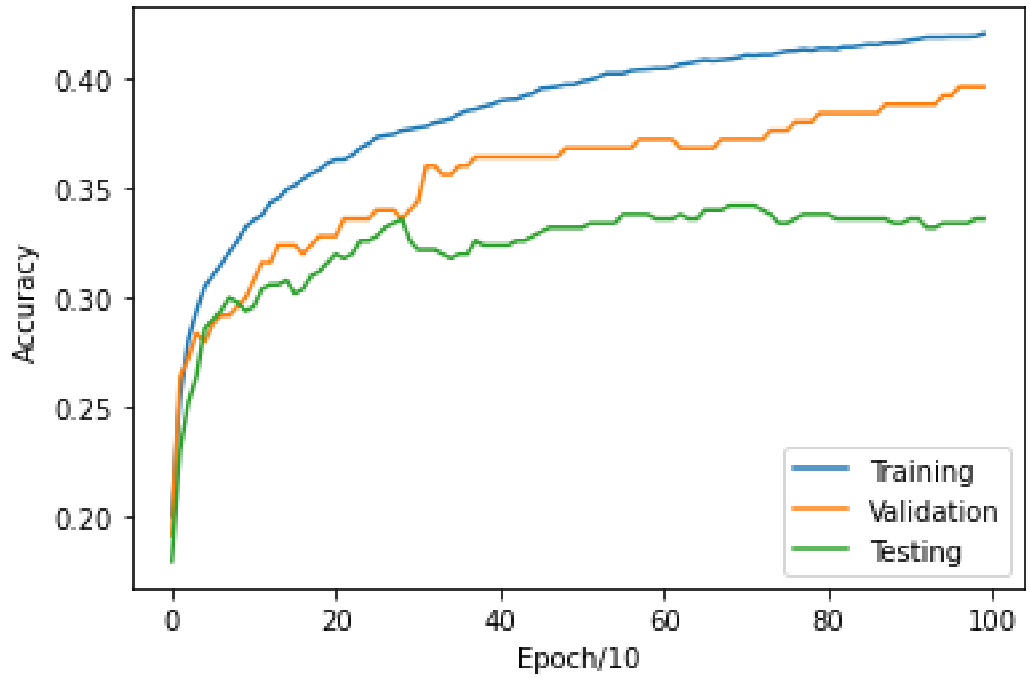
[8]:
```
# Initialize a non-zero weight_decay (Regulzarization constant) term and repeat␣
↪the training and evaluation
# Use the best learning rate as obtained from the above excercise, best_lr

# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your weight decay experiments
weight_decays = [0.00005,0.00002,0.00001]
learning_rate = 0.01

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

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

for weight_decay in weight_decays:
    # TODO: Train the classifier with different weight decay and plot
    t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
    plot_accuracies(t_ac, v_ac, te_ac)
    print(t_ac[-1], v_ac[-1], te_ac[-1])
```



0.4202 0.396 0.336



0.4204 0.396 0.336




```
0.4202 0.396 0.336
```

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

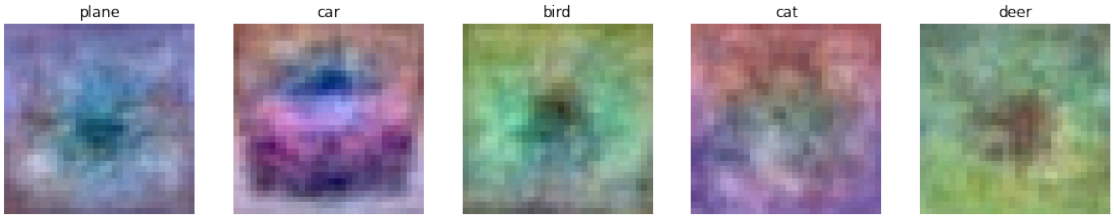
Your Answer: Underfitting occurs when the model is too simple, while overfitting occurs when it is too complex. All the weight_decay terms here gives the same classifier performance as it balances the trade-off between underfitting and overfitting, leading to good generalization performance on testing data.

2.0.4 Visualize the filters (10%)

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

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

best_learning_rate = 0.001
best_weight_decay = 0.00002
t_ac, v_ac, te_ac, best_weights = train(learning_rate, weight_decay)
```

```python
# NOTE: You need to set `best_learning_rate` and `best_weight_decay` to the␣
↪values that gave the highest accuracy
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 accuracy
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: 0.001
Best Weight Decay: 2e-05
```



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: Logistic regression performs better than the linear regression. For classification, mostly always Logistic regression better performs the linear regression. For working on a multiclass classification project, logistic regression is likely the better choice due to its ability to output probability scores and handle non-linear relationships.