knn

April 27, 2023

1 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.

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

1.1 Prepare Dataset

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

[2]:
```
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,)
```

1.2 Implementation (60%)

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

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

- Two Loop Version: use one loop to iterate through training samples and one loop to iterate through test samples

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

Note: It is possible to build a Fully Vectorized Version without explicit for loop to calculate the distance, but you do not have to do it in this assignment. You could use the fully vectorized version to replace the loop versions as well.

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

[3]:
```
from ece285.algorithms import KNN

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

1.2.1 Compare the time consumption of different method

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

[4]:
```
from ece285.utils.evaluation import get_classification_accuracy
```

Two Loop Version:

[5]:
```
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: 51.88799428939819
Test Accuracy: 0.278
```

One Loop Version

[6]:
```
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: 57.39563202857971
Test Accuracy: 0.278
```

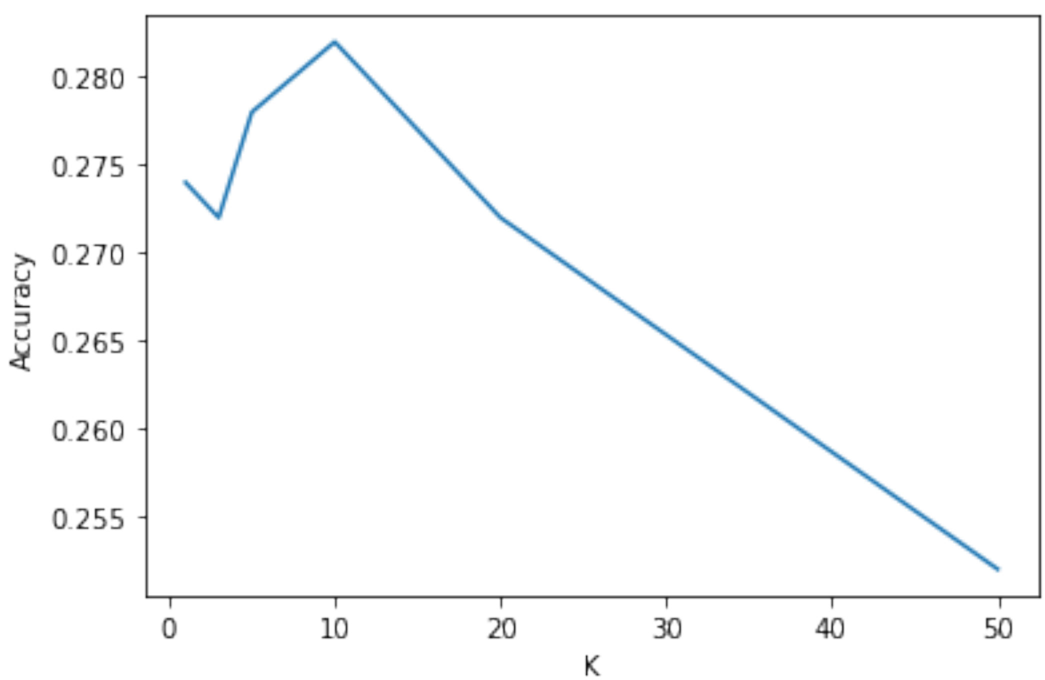
Your different implementation should output the exact same result

1.3 Test different Hyper-parameter(20%)

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

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

[7]:
```python
accuracies = []

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



1.3.1 Inline Question 1:

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

1.3.2 Your Answer:

The accuracy is highest for 10 near neighbours. When k is too low, the model may overfit the training data and be sensitive to noise in the data. On the other hand, when k is too high, the model may underfit the training data and not capture the underlying patterns in the data. So in our case k = 10 is a better estimate of the model.

1.4 Try different feature representation(20%)

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

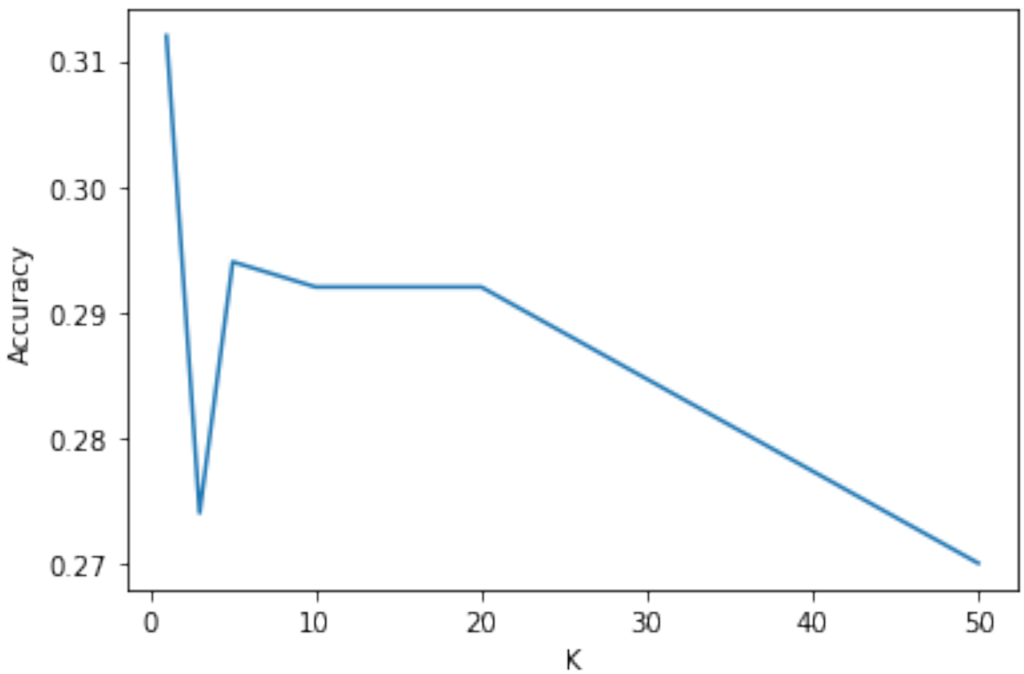
You are provided the code about using **HOG** descriptor to represent samples in the notebook.

[8]:
```
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: 118.31940770149231
```

[9]:
```
knn = KNN(num_class=10)
knn.train(
    x_train=dataset["x_train"],
    y_train=dataset["y_train"],
    k=5,
)
accuracies = []

k_candidates = [1, 3, 5, 10, 20, 50]
for k_cand in k_candidates:
```

```
    prediction = knn.predict(x_test=dataset["x_test"], k=k_cand)
    acc = get_classification_accuracy(prediction, dataset["y_test"])
    accuracies.append(acc)

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



1.4.1 Inline Question 2:

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

1.4.2 Your Answer:

The accuracy is higher when the value of k is 1, and then it has a sharp decrerase at 3. The accuracy increases a little as the k value increases to 10, and thereof has a gradual decrease. But on a whole, as compared to the previous model, the model using HOG descriptor to represent samples does a better performance in predicting classes. HOG uses a feature extraction and possible reduces the redundant features in the data, thereof leading to better prediction.

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 [14]:

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

from ece285.utils.data_processing import get_cifar10_data

# Use a subset of CIFAR10 for 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 [15]:

```
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 [16]:

```
# 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=Fa
        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
)
```

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 [17]:

```
# Import the algorithm implementation (TODO: Complete the Linear Re
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 wit
num_epochs_total = 1000  # Total number of epochs to train the clas
epochs_per_evaluation = 10  # Epochs per step of evaluation; We wil
N, D = dataset[
    "x_train"
].shape  # Get training data shape, N: Number of examples, D:Dimens
weight_decay = 0.0

# Insert additional scalar term 1 in the samples to account for the
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 [18]:

```
# 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_
    )

    # 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_

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

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

    return train_accuracies, val_accuracies, test_accuracies, weigh
```

Plot the Accuracies vs epoch graphs

In [19]:

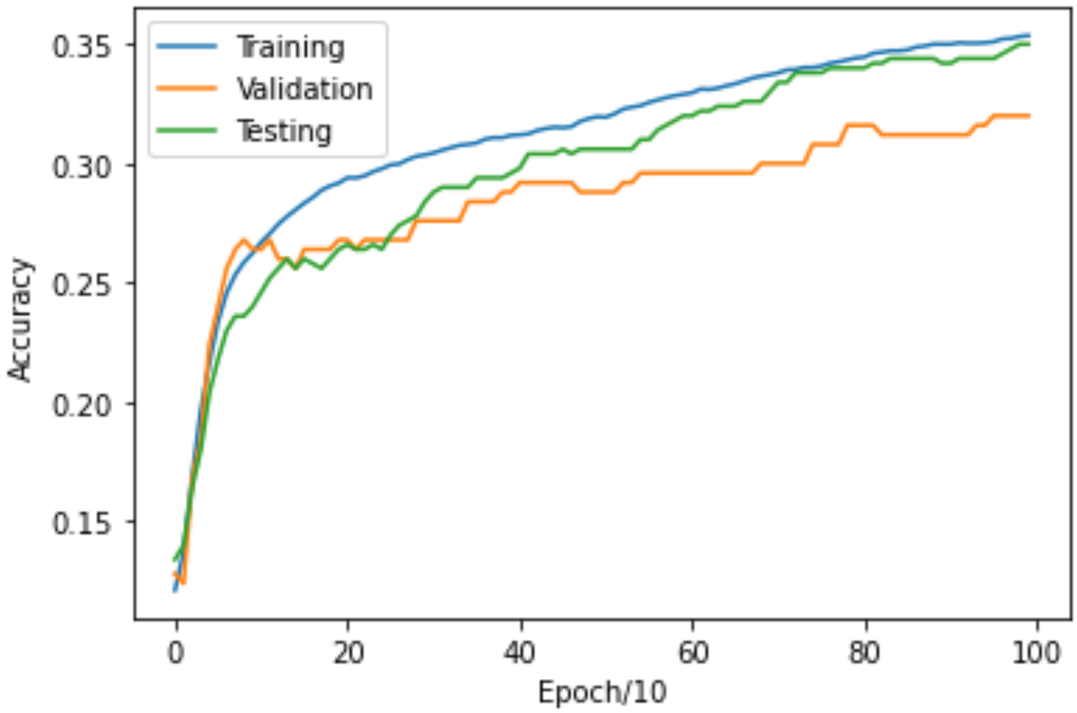
```
import matplotlib.pyplot as plt


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_evaluat
    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 [20]:

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

In [21]:

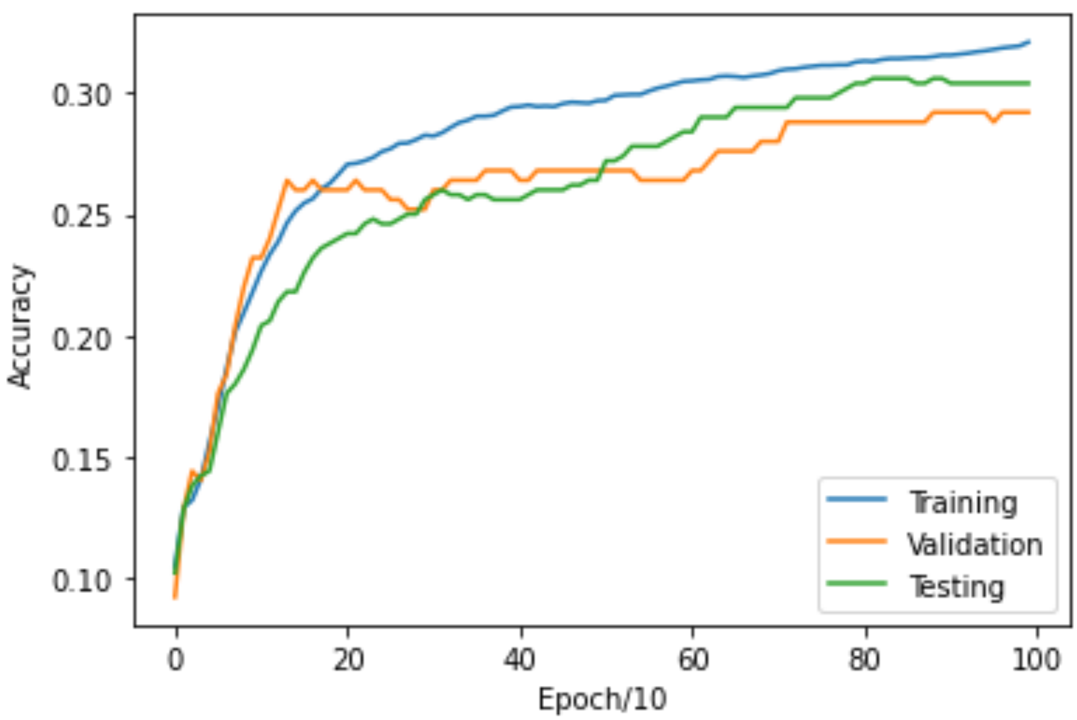
```
plot_accuracies(t_ac, v_ac, te_ac)
```

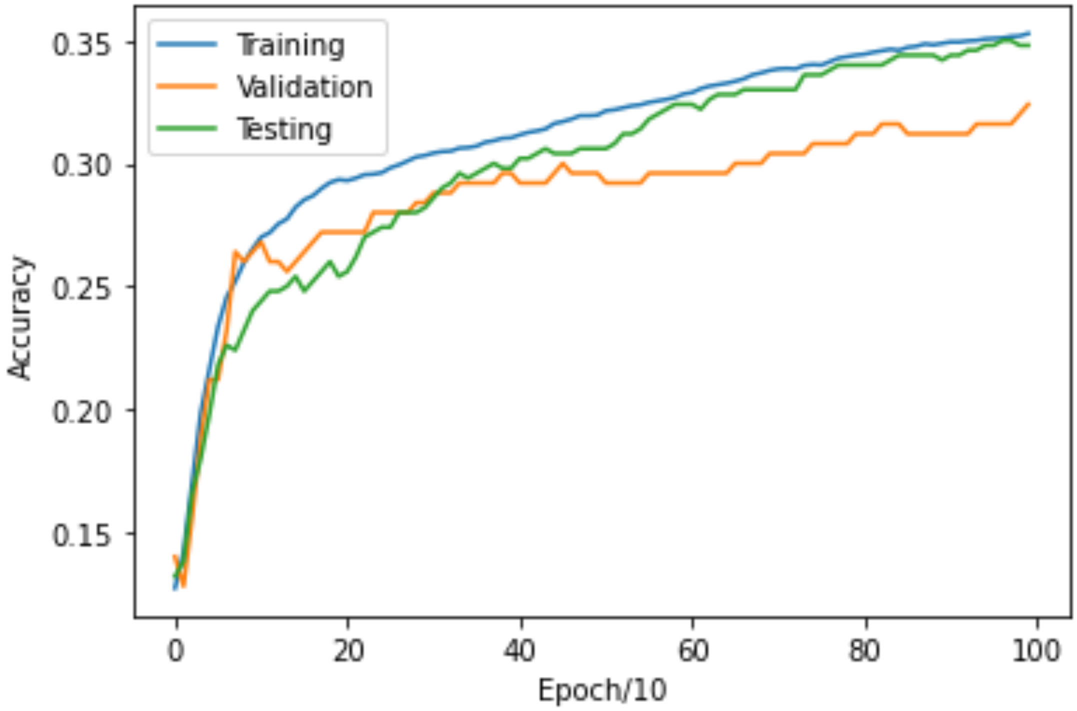


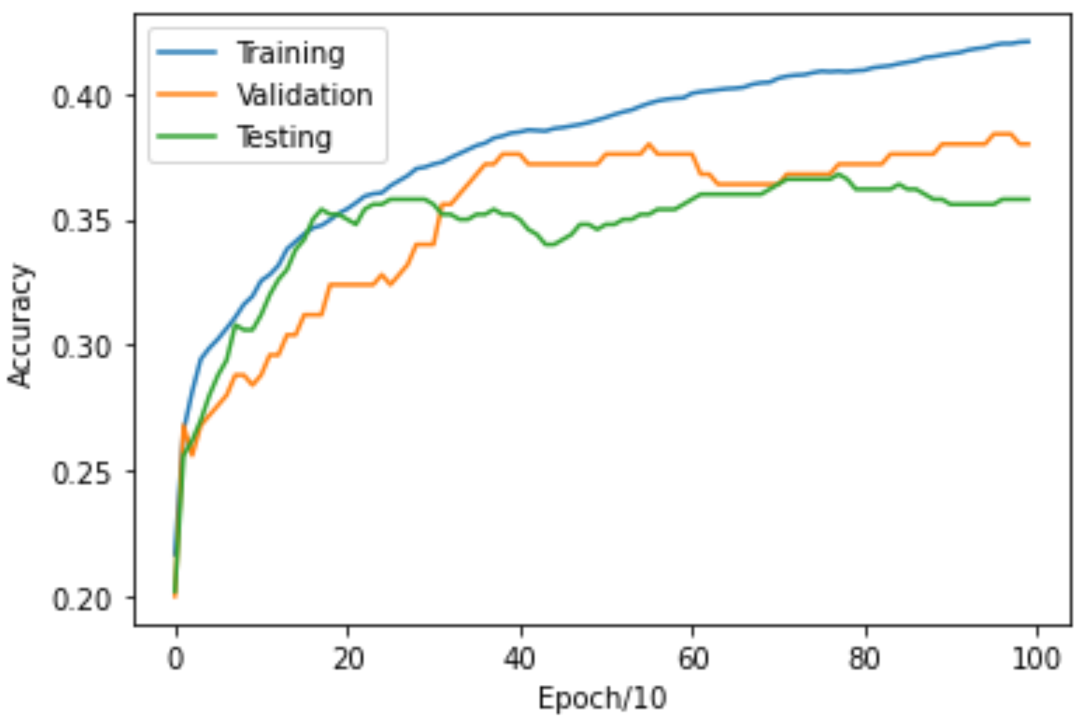
Try different learning rates and plot graphs for all (20%)

In [22]:

```
# 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
# You need to try 3 learning rates and submit all 3 graphs along wi
learning_rates = [0.00005, 0.0001, 0.0005]
weight_decay = 0.0  # No regularization for now

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

# 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
    t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
    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.

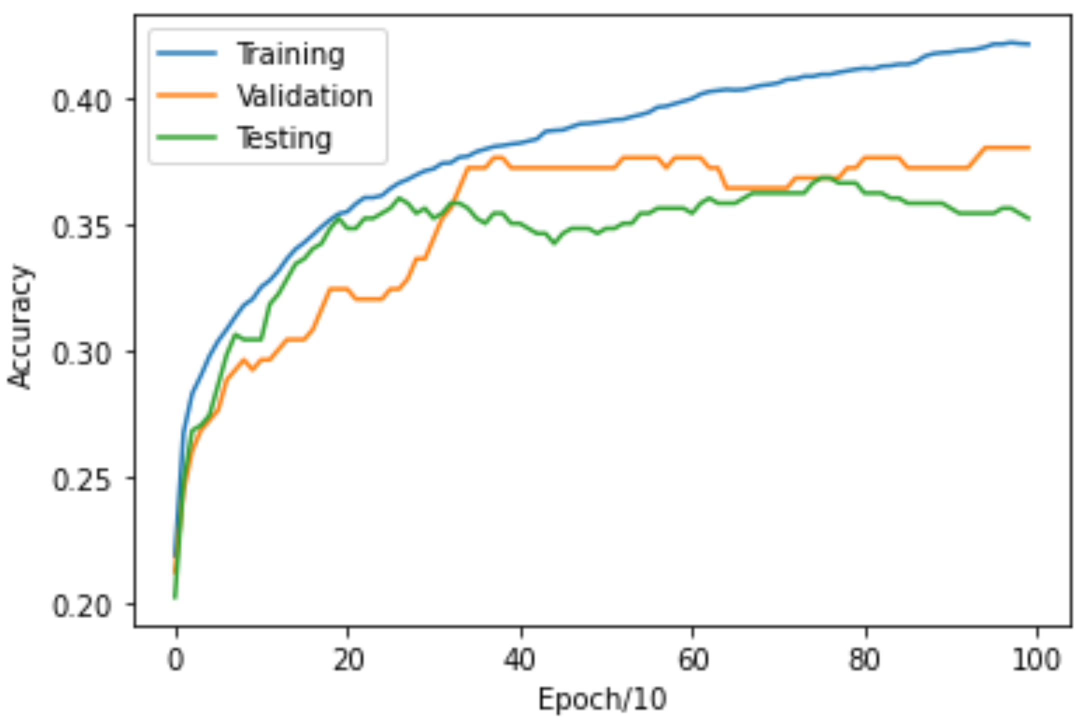
Your Answer:

The best learning rate out of the above code is 0.0005, 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.

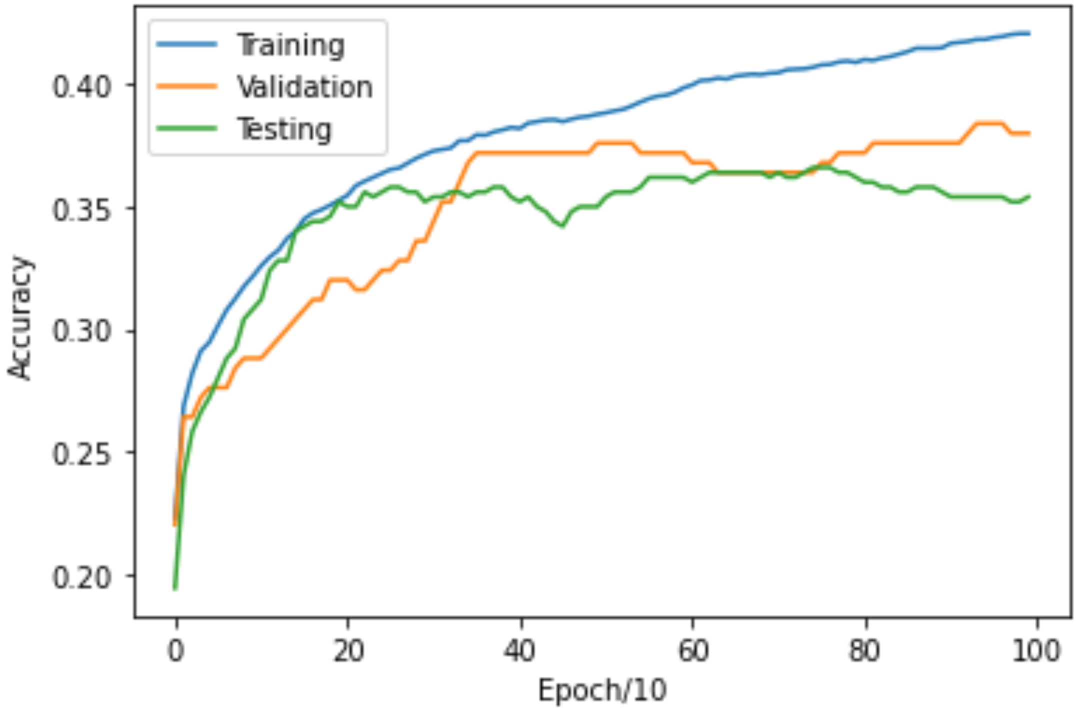
Regularization: Try different weight decay and plot graphs for all (20%)

In [23]:

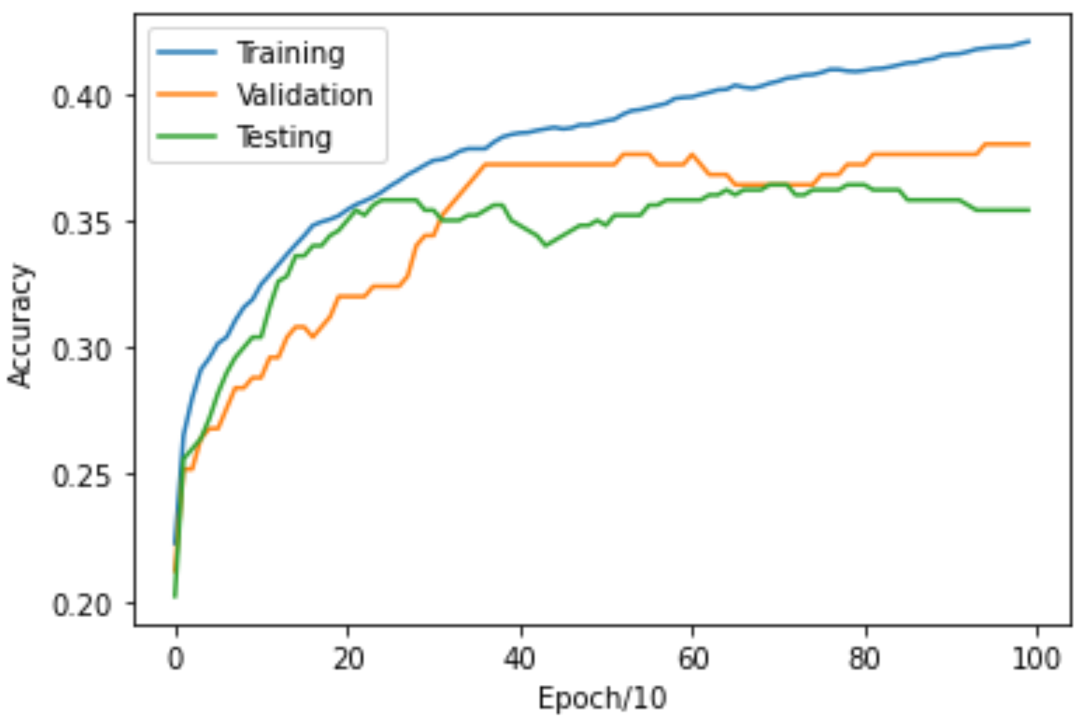
```
# Initialize a non-zero weight_decay (Regularization constant) term
# Use the best learning rate as obtained from the above exercise, b

# You need to try 3 learning rates and submit all 3 graphs along wi
weight_decays = [0.001,0.01,0.1]

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

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

for weight_decay in weight_decays:
    # TODO: Train the classifier with different weighty decay and p
    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])
```




```
0.421 0.38 0.352
```



0.4208 0.38 0.354



0.4204 0.38 0.354

Inline Question 2.

Your Answer:

Underfitting occurs when the model is too simple, while overfitting occurs when it is too complex. The weight_decay term of 0.1 gives the best classifier performance as it balances the trade-off between underfitting and overfitting, leading to good generalization performance on testing data.

Visualize the filters (10%)

In [25]:

```
# These visualizations will only somewhat make sense if your learni
# properly chosen in the model. Do your best.

# TODO: Run this cell and Show filter visualizations for the best s
# Report the 2 hyperparameters you used to obtain the best model.
best_learning_rate = 0.0005
best_weight_decay = 0.1
t_ac, v_ac, te_ac, best_weights = train(best_learning_rate, best_we

# NOTE: You need to set `best_learning_rate` and `best_weight_decay
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 high
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=(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_m
    # plt.imshow(wimg.astype('uint8'))
    plt.imshow(wimg.astype(int))
    plt.axis("off")
    plt.title(classes[i])
plt.show()
```

```
Best LR: 0.0005
Best Weight Decay: 0.1
```

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)
```

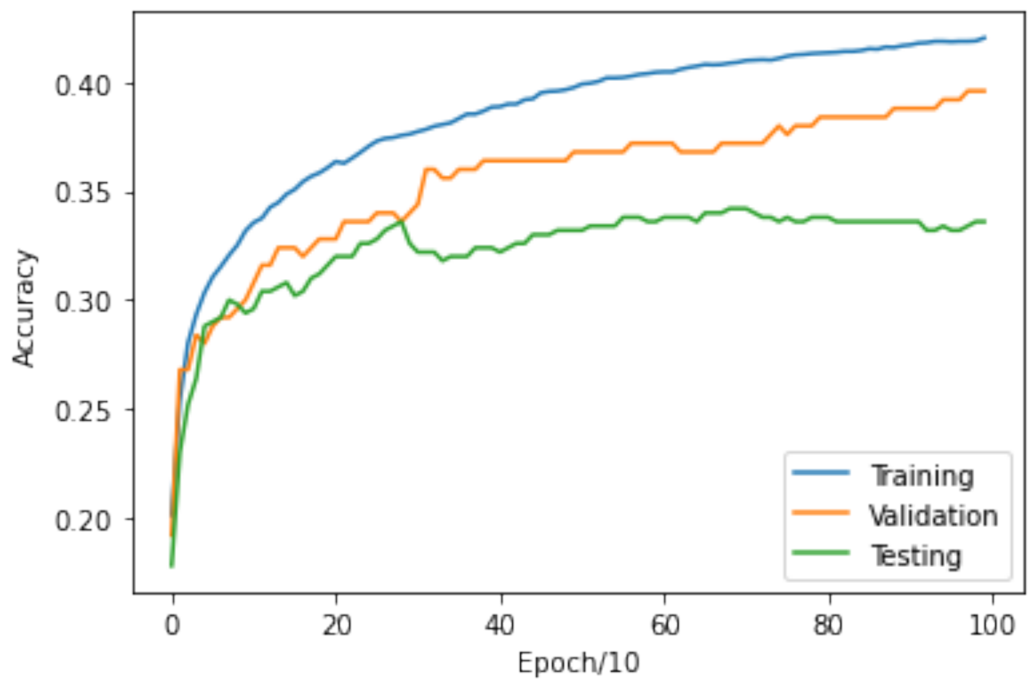
[3]:
```
# Training and evaluation function -> Outputs accuracy data
def train(learning_rate_, weight_decay_):
    # Create a linear regression object
    logistic_regression = Logistic(
        num_classes, learning_rate_, epochs_per_evaluation, weight_decay_
    )

    # 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)
```

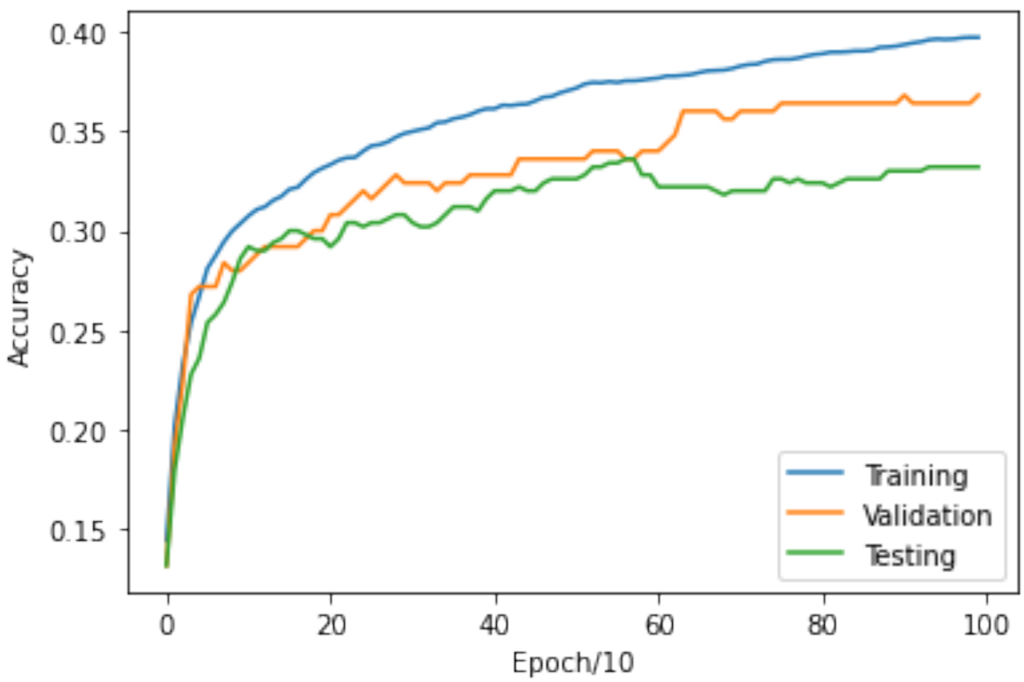
[ ]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```



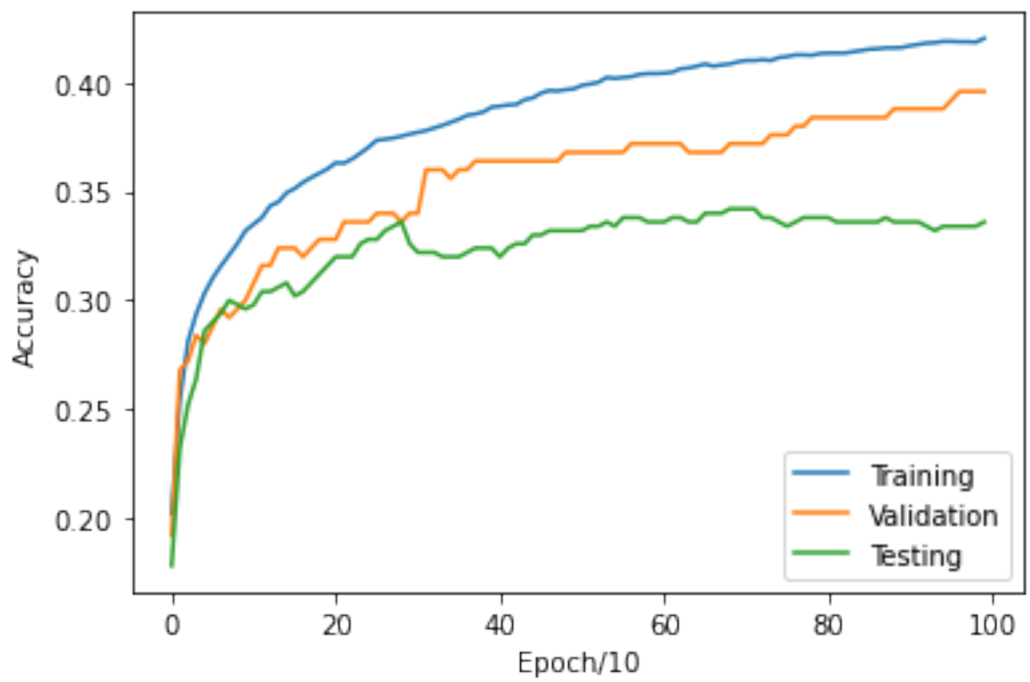
2.0.2 Try different learning rates and plot graphs for all (20%)

[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
```

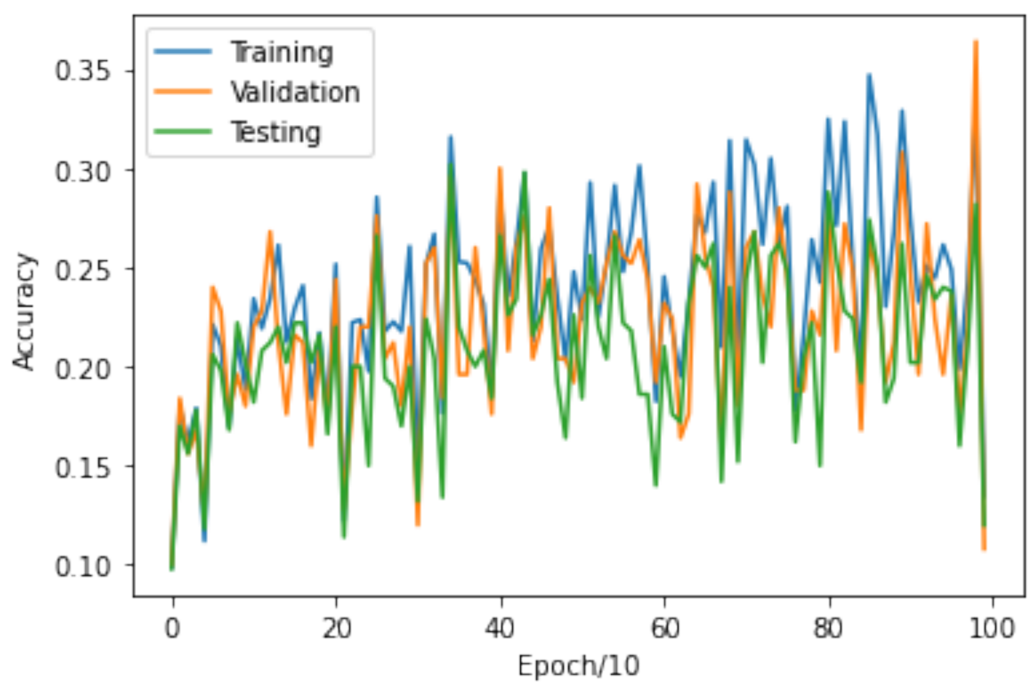
```
# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your learning rate experiments
learning_rates = [0.005, 0.01, 0.1]
weight_decay = 0.0  # No regularization for now

# 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



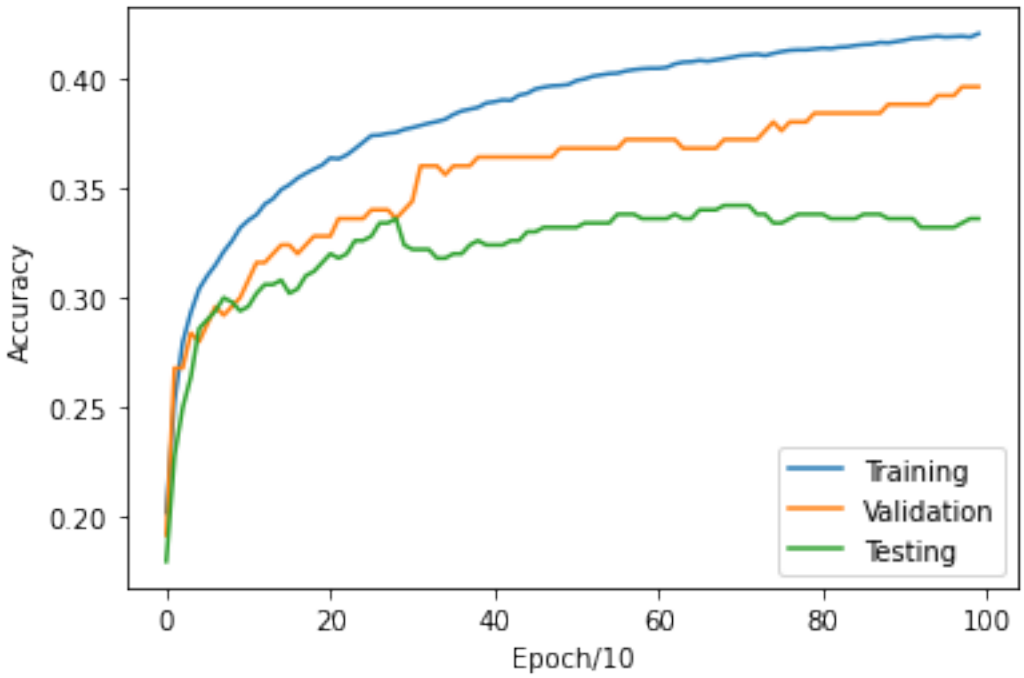
0.1344 0.108 0.12

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

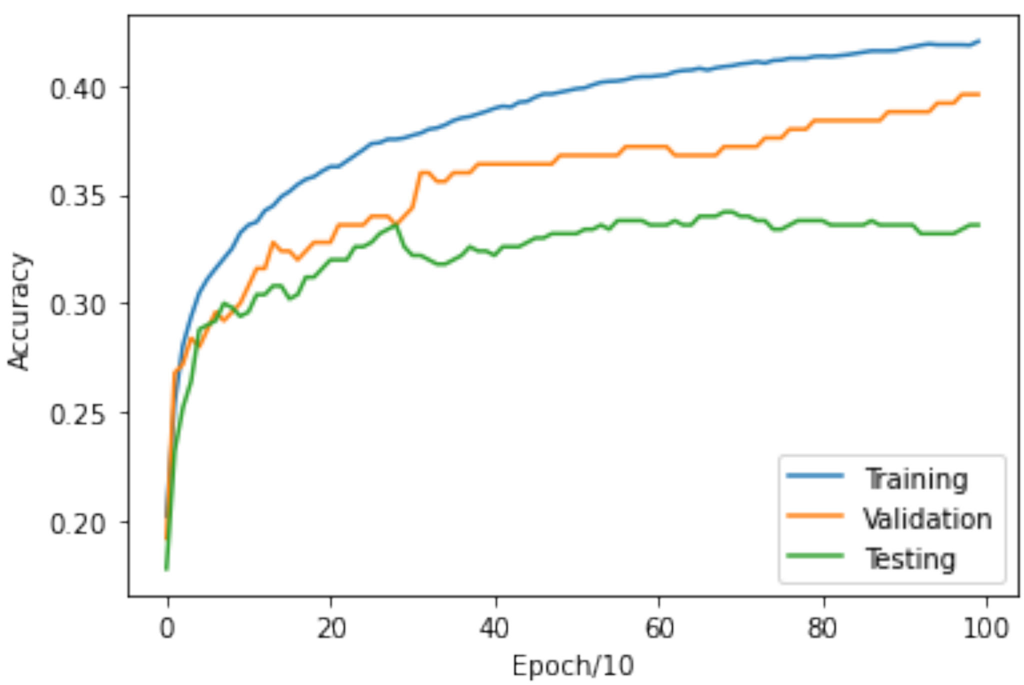
Your Answer: The best learning rate out of the above code is 0.01, and hence that would be a better pick to train the model. If the learning rate is too small, the model might converge very slowly, leading to a longer training time and less efficient optimization. Conversely, if the learning rate is 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%)

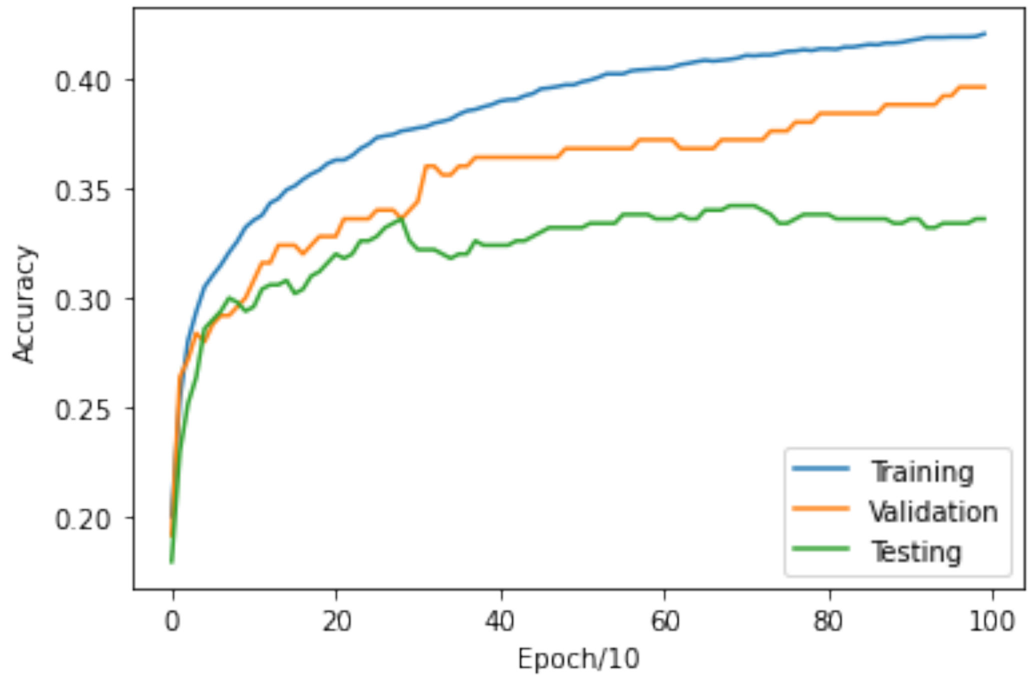
[8]:
```
# Initialize a non-zero weight_decay (Regulzarization constant) term and repeat␣
↪the training and evaluation
# Use the best learning rate as obtained from the above excercise, best_lr

# 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

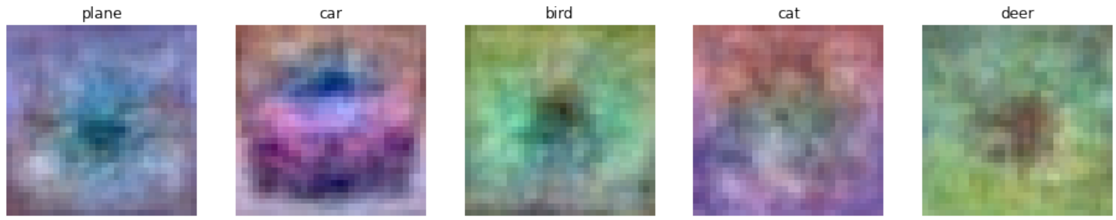
Your Answer: Underfitting occurs when the model is too simple, while overfitting occurs when it is too complex. All the weight_decay terms here gives the same classifier performance as it balances the trade-off between underfitting and overfitting, leading to good generalization performance on testing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

```
# 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.

neural_network

April 28, 2023

1 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.

[1]:
```
# 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
```

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.

[2]:
```python
# 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()
```

1.0.1 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

[3]:
```python
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
```

1.0.2 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.

[4]:
```
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)))
```

```
1.0986127233625778
Difference between your loss and correct loss:
2.220446049250313e-16
```

[5]:
```
y
```

[5]:
```
array([2, 1, 0, 1, 2, 0, 0, 2, 0, 0])
```

1.0.3 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.

We check the values for these gradients by calculating the difference, it is expected to get difference < 1e-8.

[6]:
```
# 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,)
```

[7]:
```
# No need to edit anything in this block ( 20% of the above 40% )
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 7.701916434367482e-09
```

[8]:
```
gradients[3].shape
```

[8]:
```
(3,)
```

1.1 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.

[9]:
```
# 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 sequentially to
# get the gradients and optimize the respective parameters.
# You should work with the given parameters and only edit your implementation␣
↪in the .py files

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 gradients
    optim.step()
    # Append the loss at each iteration
    epoch_loss.append(loss)

    if (epoch + 1) % 50 == 0:
```

```
        print("Epoch {}, loss={:3f}".format(epoch + 1, epoch_loss[-1]))
```

```
Epoch 50, loss=0.832706
Epoch 100, loss=0.454687
Epoch 150, loss=0.118350
Epoch 200, loss=0.055911
Epoch 250, loss=0.038039
Epoch 300, loss=0.029528
Epoch 350, loss=0.024400
Epoch 400, loss=0.020819
Epoch 450, loss=0.017947
Epoch 500, loss=0.015866
Epoch 550, loss=0.014198
Epoch 600, loss=0.012916
Epoch 650, loss=0.011859
Epoch 700, loss=0.010943
Epoch 750, loss=0.010198
Epoch 800, loss=0.009540
Epoch 850, loss=0.008970
Epoch 900, loss=0.008454
Epoch 950, loss=0.008003
Epoch 1000, loss=0.007593
```

[10]:
```
# Test your predictions. The predictions must match the labels
print(net.predict(X))
print(y)
```

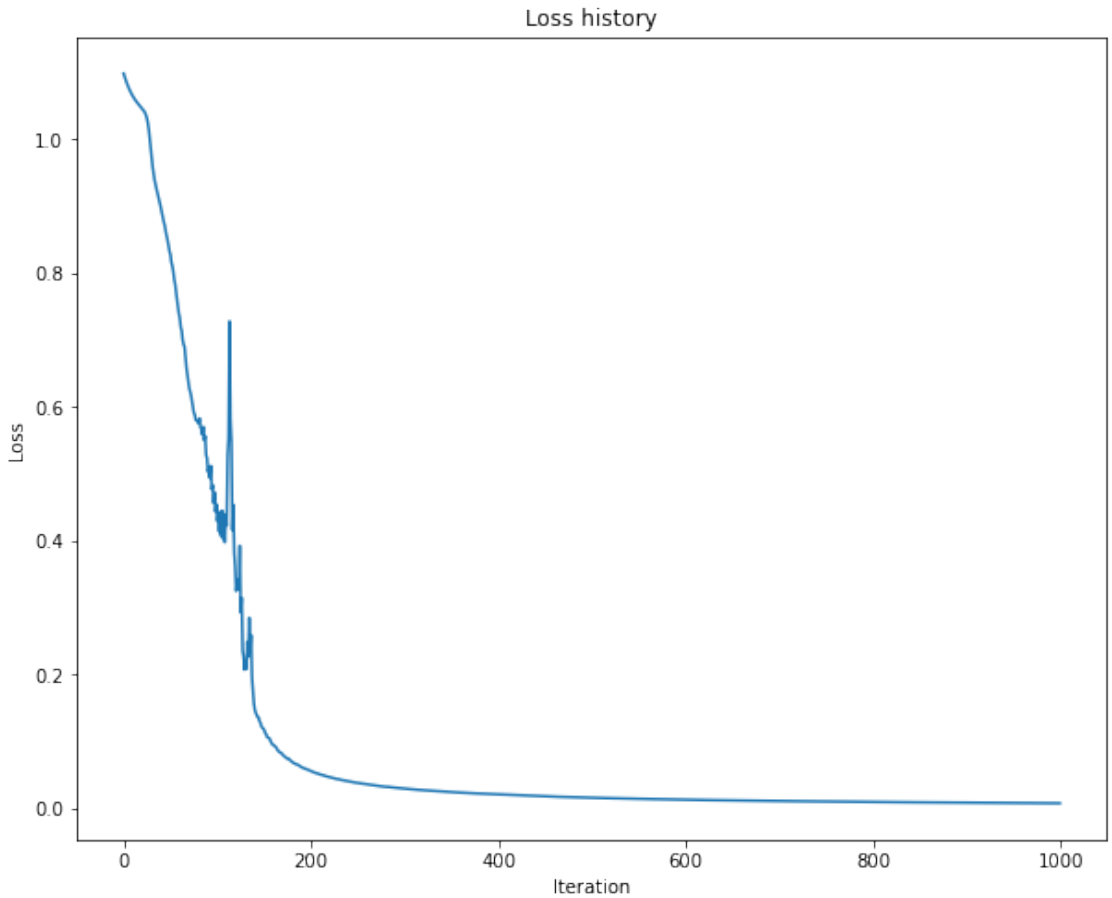
```
[2 1 0 1 2 0 0 2 0 0]
[2 1 0 1 2 0 0 2 0 0]
```

[11]:
```
# 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 0.007593419801731282
```

[12]:
```
# 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")
```

[12]:
```
Text(0, 0.5, 'Loss')
```


Loss history
Loss
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
Iteration

classification_nn

April 28, 2023

1 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.

[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,)
```

[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"]
```

[3]:
```
# 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
```

1.1 Visualize some examples from the dataset.

[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
```

```python
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,
)
```

1.2 Initialize the model

[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␣
↪Approximation Theorem.
# A 2 layer neural network with non-linearities can approximate any function,␣
↪given large enough hidden layer
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])
```

[6]:
```
# 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)
```

[7]:
```
# Initialize the trainer class by passing the above modules
trainer = Trainer(
    dataset, optim, net, loss_func, epoch, batch_size, validate_interval=3
)
```

[8]:
```
# Call the trainer function we have already implemented for you. This trains␣
↪the model for the given
# hyper-parameters. It follows the same procedure as in the last ipython␣
↪notebook you used for the toy-dataset
train_error, validation_accuracy = trainer.train()
```

```
Epoch Average Loss: 2.302540
Validate Acc: 0.084
Epoch Average Loss: 2.302361
Epoch Average Loss: 2.302151
Epoch Average Loss: 2.301853
Validate Acc: 0.104
Epoch Average Loss: 2.301432
Epoch Average Loss: 2.300826
Epoch Average Loss: 2.299973
Validate Acc: 0.092
Epoch Average Loss: 2.298809
Epoch Average Loss: 2.297303
Epoch Average Loss: 2.295474
Validate Acc: 0.084
Epoch Average Loss: 2.293317
Epoch Average Loss: 2.290796
Epoch Average Loss: 2.287742
Validate Acc: 0.084
Epoch Average Loss: 2.283805
Epoch Average Loss: 2.278752
Epoch Average Loss: 2.272493
Validate Acc: 0.096
Epoch Average Loss: 2.265579
Epoch Average Loss: 2.258138
Epoch Average Loss: 2.250392
```

```
Validate Acc: 0.108
Epoch Average Loss: 2.242758
Epoch Average Loss: 2.235319
Epoch Average Loss: 2.228368
Validate Acc: 0.112
Epoch Average Loss: 2.221744
Epoch Average Loss: 2.215524
Epoch Average Loss: 2.209779
Validate Acc: 0.124
Epoch Average Loss: 2.204544
Epoch Average Loss: 2.199672
Epoch Average Loss: 2.195220
Validate Acc: 0.136
Epoch Average Loss: 2.191033
Epoch Average Loss: 2.187090
Epoch Average Loss: 2.183473
Validate Acc: 0.140
Epoch Average Loss: 2.179802
Epoch Average Loss: 2.176372
Epoch Average Loss: 2.173139
Validate Acc: 0.140
Epoch Average Loss: 2.170171
Epoch Average Loss: 2.167052
Epoch Average Loss: 2.164485
Validate Acc: 0.140
Epoch Average Loss: 2.161738
Epoch Average Loss: 2.159202
Epoch Average Loss: 2.156774
Validate Acc: 0.144
Epoch Average Loss: 2.154426
Epoch Average Loss: 2.151992
Epoch Average Loss: 2.149871
Validate Acc: 0.148
Epoch Average Loss: 2.148015
Epoch Average Loss: 2.145909
Epoch Average Loss: 2.143988
Validate Acc: 0.148
Epoch Average Loss: 2.142119
Epoch Average Loss: 2.140410
Epoch Average Loss: 2.138593
Validate Acc: 0.148
Epoch Average Loss: 2.136911
Epoch Average Loss: 2.134978
Epoch Average Loss: 2.133238
Validate Acc: 0.152
Epoch Average Loss: 2.132031
Epoch Average Loss: 2.130097
Epoch Average Loss: 2.128498
```

```
Validate Acc: 0.152
Epoch Average Loss: 2.127047
Epoch Average Loss: 2.125789
Epoch Average Loss: 2.123959
Validate Acc: 0.164
Epoch Average Loss: 2.122555
Epoch Average Loss: 2.120989
Epoch Average Loss: 2.119210
Validate Acc: 0.168
Epoch Average Loss: 2.117758
Epoch Average Loss: 2.116090
Epoch Average Loss: 2.114072
Validate Acc: 0.160
Epoch Average Loss: 2.112825
Epoch Average Loss: 2.111065
Epoch Average Loss: 2.109000
Validate Acc: 0.172
Epoch Average Loss: 2.106939
Epoch Average Loss: 2.104999
Epoch Average Loss: 2.102819
Validate Acc: 0.172
Epoch Average Loss: 2.100567
Epoch Average Loss: 2.098402
Epoch Average Loss: 2.095404
Validate Acc: 0.176
Epoch Average Loss: 2.093283
Epoch Average Loss: 2.090502
Epoch Average Loss: 2.087675
Validate Acc: 0.188
Epoch Average Loss: 2.084531
Epoch Average Loss: 2.081617
Epoch Average Loss: 2.078510
Validate Acc: 0.228
Epoch Average Loss: 2.075273
Epoch Average Loss: 2.072233
Epoch Average Loss: 2.069277
Validate Acc: 0.220
Epoch Average Loss: 2.065468
Epoch Average Loss: 2.062220
Epoch Average Loss: 2.059001
Validate Acc: 0.224
Epoch Average Loss: 2.056238
Epoch Average Loss: 2.053032
Epoch Average Loss: 2.050105
Validate Acc: 0.244
Epoch Average Loss: 2.047188
Epoch Average Loss: 2.044075
Epoch Average Loss: 2.041257
```

```
Validate Acc: 0.248
Epoch Average Loss: 2.038873
Epoch Average Loss: 2.035811
Epoch Average Loss: 2.032747
Validate Acc: 0.240
Epoch Average Loss: 2.030543
Epoch Average Loss: 2.028426
Epoch Average Loss: 2.025668
Validate Acc: 0.268
Epoch Average Loss: 2.023039
Epoch Average Loss: 2.020892
Epoch Average Loss: 2.019254
Validate Acc: 0.260
Epoch Average Loss: 2.016086
Epoch Average Loss: 2.014286
Epoch Average Loss: 2.011895
Validate Acc: 0.268
Epoch Average Loss: 2.009652
Epoch Average Loss: 2.008635
Epoch Average Loss: 2.006483
Validate Acc: 0.272
Epoch Average Loss: 2.004348
Epoch Average Loss: 2.001680
Epoch Average Loss: 2.000448
Validate Acc: 0.264
Epoch Average Loss: 1.998511
Epoch Average Loss: 1.996869
Epoch Average Loss: 1.994683
Validate Acc: 0.272
Epoch Average Loss: 1.992685
Epoch Average Loss: 1.990515
Epoch Average Loss: 1.989023
Validate Acc: 0.280
Epoch Average Loss: 1.987115
Epoch Average Loss: 1.984757
Epoch Average Loss: 1.983362
Validate Acc: 0.276
Epoch Average Loss: 1.980770
Epoch Average Loss: 1.978400
Epoch Average Loss: 1.976779
Validate Acc: 0.288
Epoch Average Loss: 1.974880
Epoch Average Loss: 1.973159
Epoch Average Loss: 1.970188
Validate Acc: 0.292
Epoch Average Loss: 1.967808
Epoch Average Loss: 1.965204
Epoch Average Loss: 1.962598
```

```
Validate Acc: 0.292
Epoch Average Loss: 1.959931
Epoch Average Loss: 1.956976
Epoch Average Loss: 1.954877
Validate Acc: 0.284
Epoch Average Loss: 1.951932
Epoch Average Loss: 1.948408
Epoch Average Loss: 1.944597
Validate Acc: 0.296
Epoch Average Loss: 1.943494
Epoch Average Loss: 1.939637
Epoch Average Loss: 1.937788
Validate Acc: 0.296
Epoch Average Loss: 1.934564
Epoch Average Loss: 1.931541
Epoch Average Loss: 1.930091
Validate Acc: 0.268
Epoch Average Loss: 1.926533
Epoch Average Loss: 1.925099
Epoch Average Loss: 1.922507
Validate Acc: 0.272
Epoch Average Loss: 1.920127
Epoch Average Loss: 1.918304
Epoch Average Loss: 1.916498
Validate Acc: 0.288
Epoch Average Loss: 1.913333
Epoch Average Loss: 1.911053
Epoch Average Loss: 1.908856
Validate Acc: 0.284
Epoch Average Loss: 1.907668
Epoch Average Loss: 1.905090
Epoch Average Loss: 1.902684
Validate Acc: 0.280
Epoch Average Loss: 1.900746
Epoch Average Loss: 1.898557
Epoch Average Loss: 1.897128
Validate Acc: 0.308
Epoch Average Loss: 1.894018
Validate Acc: 0.308
Epoch Average Loss: 1.887973
Epoch Average Loss: 1.887830
Epoch Average Loss: 1.885539
Validate Acc: 0.308
Epoch Average Loss: 1.883132
Epoch Average Loss: 1.879233
Epoch Average Loss: 1.880205
Validate Acc: 0.300
Epoch Average Loss: 1.875765
```

```
Epoch Average Loss: 1.874388
Epoch Average Loss: 1.872908
Validate Acc: 0.292
Epoch Average Loss: 1.869481
Epoch Average Loss: 1.868074
Epoch Average Loss: 1.867479
Validate Acc: 0.300
Epoch Average Loss: 1.863720
Epoch Average Loss: 1.862010
Epoch Average Loss: 1.860678
Validate Acc: 0.300
Epoch Average Loss: 1.859408
Epoch Average Loss: 1.857180
Epoch Average Loss: 1.854840
Validate Acc: 0.300
Epoch Average Loss: 1.853317
Epoch Average Loss: 1.850804
Epoch Average Loss: 1.849211
Validate Acc: 0.304
Epoch Average Loss: 1.847043
Epoch Average Loss: 1.846299
Epoch Average Loss: 1.842838
Validate Acc: 0.316
Epoch Average Loss: 1.841435
Epoch Average Loss: 1.840141
Epoch Average Loss: 1.837533
Validate Acc: 0.304
Epoch Average Loss: 1.837839
Epoch Average Loss: 1.835047
Epoch Average Loss: 1.832190
Validate Acc: 0.324
Epoch Average Loss: 1.830337
Epoch Average Loss: 1.828371
Epoch Average Loss: 1.826309
Validate Acc: 0.312
Epoch Average Loss: 1.824816
Epoch Average Loss: 1.822740
Epoch Average Loss: 1.822450
Validate Acc: 0.316
Epoch Average Loss: 1.818955
Epoch Average Loss: 1.816644
Epoch Average Loss: 1.815093
Validate Acc: 0.312
Epoch Average Loss: 1.812639
Epoch Average Loss: 1.811698
Epoch Average Loss: 1.810282
Validate Acc: 0.320
Epoch Average Loss: 1.807828
```

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

[9]:
```
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:  0.3476
Validation acc:  0.328
```

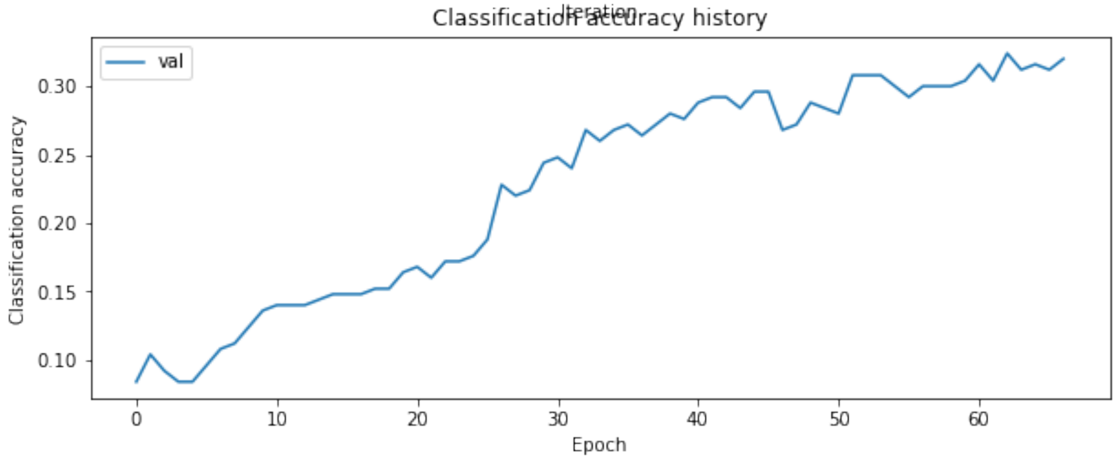
1.2.2 Debug the training

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.

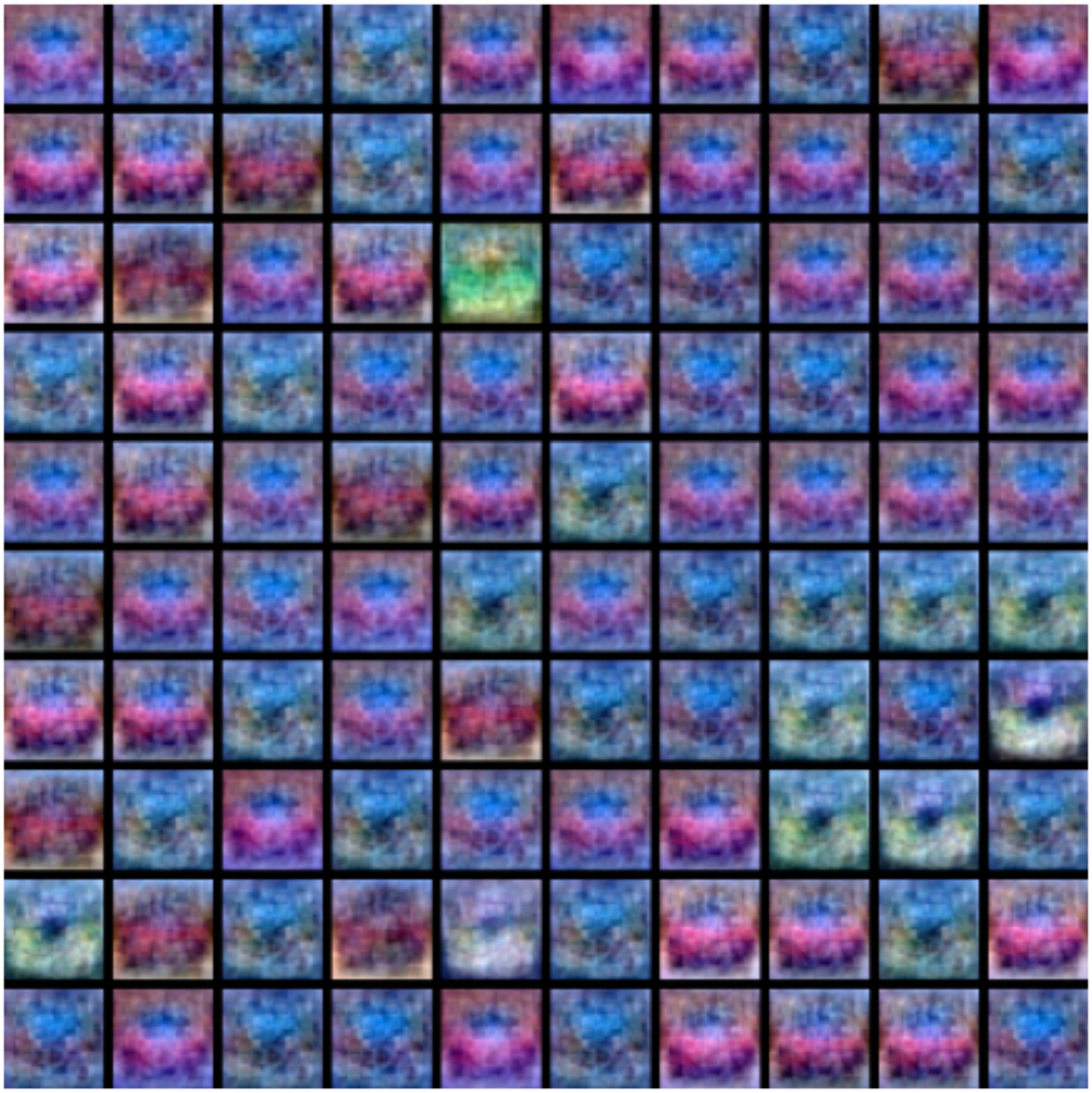
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[10]:
```
# 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()
```



[7]:
```
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()
```

[11]:
```
show_net_weights(net)
```



2 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: I have taken three values for each parameters and run them seperately. lr = [0.005,0.01, 0.1] weight_decay = [0.01, 0.001, 0.005] epoch = [200, 250, 300] hidden_size = [200, 220, 250] Out of which, I chose the hyperparameters for our model. The best parameters is for the following configuration of hyperparameters: lr = 0.01 weight_decay = 0.005 epoch = 300 hidden_size = 220

I have given below a three such iterations I had done as example.

[6]:
```
best_net_hyperparams = None  # 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␣
 ↪#
```

```python
# as a list in the order
# best_net_hyperparams = [lr, weight_decay, epoch, hidden_size]
###################################################################################

lr = 0.005
weight_decay = 0.01
epoch = 300
hidden_size = 250

dataset = DataLoader(x_train, y_train, x_val, y_val, x_test, y_test)
net = init_model()
optim = SGD(net, lr, weight_decay)
loss_func = CrossEntropyLoss()
batch_size = 200  # (Reduce the batch size if your computer is unable to handle
 ↪it)

trainer = Trainer(
    dataset, optim, net, loss_func, epoch, batch_size, validate_interval=3
)


train_error, validation_accuracy = trainer.train()

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)
```

```
Epoch Average Loss: 2.302526
Validate Acc: 0.100
Epoch Average Loss: 2.302349
Epoch Average Loss: 2.302157
Epoch Average Loss: 2.301955
Validate Acc: 0.100
Epoch Average Loss: 2.301731
Epoch Average Loss: 2.301477
Epoch Average Loss: 2.301186
Validate Acc: 0.096
Epoch Average Loss: 2.300845
Epoch Average Loss: 2.300439
Epoch Average Loss: 2.299985
Validate Acc: 0.100
Epoch Average Loss: 2.299462
Epoch Average Loss: 2.298887
Epoch Average Loss: 2.298226
```

```
Validate Acc: 0.100
Epoch Average Loss: 2.295913
Validate Acc: 0.096
Epoch Average Loss: 2.295035
Epoch Average Loss: 2.294081
Epoch Average Loss: 2.293082
Validate Acc: 0.100
Epoch Average Loss: 2.291980
Epoch Average Loss: 2.290742
Epoch Average Loss: 2.289412
Validate Acc: 0.084
Epoch Average Loss: 2.287895
Epoch Average Loss: 2.286176
Epoch Average Loss: 2.284228
Validate Acc: 0.084
Epoch Average Loss: 2.281985
Epoch Average Loss: 2.279427
Epoch Average Loss: 2.276528
Validate Acc: 0.096
Epoch Average Loss: 2.273350
Epoch Average Loss: 2.269966
Epoch Average Loss: 2.266377
Validate Acc: 0.096
Epoch Average Loss: 2.262713
Epoch Average Loss: 2.258927
Epoch Average Loss: 2.255181
Validate Acc: 0.096
Epoch Average Loss: 2.251291
Epoch Average Loss: 2.247433
Epoch Average Loss: 2.243567
Validate Acc: 0.108
Epoch Average Loss: 2.239760
Epoch Average Loss: 2.236036
Epoch Average Loss: 2.232396
Validate Acc: 0.112
Epoch Average Loss: 2.228895
Epoch Average Loss: 2.225436
Epoch Average Loss: 2.222147
Validate Acc: 0.120
Epoch Average Loss: 2.218992
Epoch Average Loss: 2.215883
Epoch Average Loss: 2.212952
Validate Acc: 0.124
Epoch Average Loss: 2.210105
Epoch Average Loss: 2.207410
Epoch Average Loss: 2.204794
Validate Acc: 0.124
Epoch Average Loss: 2.202271
```

```
Epoch Average Loss: 2.199806
Epoch Average Loss: 2.197549
Validate Acc: 0.132
Epoch Average Loss: 2.195255
Epoch Average Loss: 2.193093
Epoch Average Loss: 2.190996
Validate Acc: 0.132
Epoch Average Loss: 2.189070
Epoch Average Loss: 2.187109
Epoch Average Loss: 2.185097
Validate Acc: 0.132
Epoch Average Loss: 2.183253
Epoch Average Loss: 2.181414
Epoch Average Loss: 2.179774
Validate Acc: 0.136
Epoch Average Loss: 2.178070
Epoch Average Loss: 2.176245
Epoch Average Loss: 2.174605
Validate Acc: 0.140
Epoch Average Loss: 2.172976
Epoch Average Loss: 2.171556
Epoch Average Loss: 2.169855
Validate Acc: 0.144
Epoch Average Loss: 2.168502
Epoch Average Loss: 2.167123
Epoch Average Loss: 2.165573
Validate Acc: 0.136
Epoch Average Loss: 2.164206
Epoch Average Loss: 2.162870
Epoch Average Loss: 2.161557
Validate Acc: 0.140
Epoch Average Loss: 2.160264
Epoch Average Loss: 2.159208
Epoch Average Loss: 2.157845
Validate Acc: 0.144
Epoch Average Loss: 2.156593
Epoch Average Loss: 2.155301
Epoch Average Loss: 2.154261
Validate Acc: 0.148
Epoch Average Loss: 2.153046
Epoch Average Loss: 2.151867
Epoch Average Loss: 2.150792
Validate Acc: 0.156
Epoch Average Loss: 2.149738
Epoch Average Loss: 2.148551
Epoch Average Loss: 2.147613
Validate Acc: 0.148
Epoch Average Loss: 2.146707
```

```
Epoch Average Loss: 2.145632
Epoch Average Loss: 2.144618
Validate Acc: 0.148
Epoch Average Loss: 2.143601
Epoch Average Loss: 2.142692
Epoch Average Loss: 2.141743
Validate Acc: 0.152
Epoch Average Loss: 2.140732
Epoch Average Loss: 2.139929
Epoch Average Loss: 2.138949
Validate Acc: 0.156
Epoch Average Loss: 2.137964
Epoch Average Loss: 2.137243
Epoch Average Loss: 2.136265
Validate Acc: 0.156
Epoch Average Loss: 2.135400
Epoch Average Loss: 2.134447
Epoch Average Loss: 2.133841
Validate Acc: 0.156
Epoch Average Loss: 2.132902
Epoch Average Loss: 2.132003
Epoch Average Loss: 2.131105
Validate Acc: 0.156
Epoch Average Loss: 2.130283
Epoch Average Loss: 2.129488
Epoch Average Loss: 2.128787
Validate Acc: 0.152
Epoch Average Loss: 2.127793
Epoch Average Loss: 2.127045
Epoch Average Loss: 2.126136
Validate Acc: 0.156
Epoch Average Loss: 2.125544
Validate Acc: 0.164
Epoch Average Loss: 2.122924
Epoch Average Loss: 2.122021
Epoch Average Loss: 2.121062
Validate Acc: 0.164
Epoch Average Loss: 2.120241
Epoch Average Loss: 2.119401
Epoch Average Loss: 2.118580
Validate Acc: 0.168
Epoch Average Loss: 2.117596
Epoch Average Loss: 2.116683
Epoch Average Loss: 2.115751
Validate Acc: 0.176
Epoch Average Loss: 2.114867
Epoch Average Loss: 2.113860
Epoch Average Loss: 2.113115
```

```
Validate Acc: 0.176
Epoch Average Loss: 2.111935
Epoch Average Loss: 2.110977
Epoch Average Loss: 2.105825
Epoch Average Loss: 2.104365
Epoch Average Loss: 2.103172
Validate Acc: 0.172
Epoch Average Loss: 2.102016
Epoch Average Loss: 2.100846
Epoch Average Loss: 2.099578
Validate Acc: 0.196
Epoch Average Loss: 2.098230
Epoch Average Loss: 2.096877
Epoch Average Loss: 2.095460
Validate Acc: 0.192
Epoch Average Loss: 2.094212
Validate Acc: 0.196
Epoch Average Loss: 2.089738
Epoch Average Loss: 2.088225
Epoch Average Loss: 2.086720
Validate Acc: 0.212
Epoch Average Loss: 2.085139
Epoch Average Loss: 2.083597
Epoch Average Loss: 2.082021
Validate Acc: 0.232
Epoch Average Loss: 2.080342
Epoch Average Loss: 2.078760
Epoch Average Loss: 2.077136
Validate Acc: 0.232
Epoch Average Loss: 2.075617
Epoch Average Loss: 2.073754
Epoch Average Loss: 2.072305
Validate Acc: 0.224
Epoch Average Loss: 2.070566
Epoch Average Loss: 2.068961
Epoch Average Loss: 2.067367
Validate Acc: 0.212
Epoch Average Loss: 2.065681
Epoch Average Loss: 2.063937
Epoch Average Loss: 2.062471
Validate Acc: 0.228
Epoch Average Loss: 2.060685
Epoch Average Loss: 2.059210
Epoch Average Loss: 2.057739
Validate Acc: 0.236
Epoch Average Loss: 2.056144
Epoch Average Loss: 2.054395
Epoch Average Loss: 2.052909
```

```
Validate Acc: 0.236
Epoch Average Loss: 2.051298
Epoch Average Loss: 2.049977
Epoch Average Loss: 2.048219
Validate Acc: 0.240
Epoch Average Loss: 2.046737
Epoch Average Loss: 2.045331
Epoch Average Loss: 2.043901
Validate Acc: 0.248
Epoch Average Loss: 2.042262
Epoch Average Loss: 2.040729
Epoch Average Loss: 2.039529
Validate Acc: 0.240
Epoch Average Loss: 2.038338
Epoch Average Loss: 2.036717
Epoch Average Loss: 2.035407
Validate Acc: 0.244
Epoch Average Loss: 2.034201
Epoch Average Loss: 2.032762
Epoch Average Loss: 2.031650
Validate Acc: 0.260
Epoch Average Loss: 2.030036
Epoch Average Loss: 2.029005
Epoch Average Loss: 2.027716
Validate Acc: 0.248
Epoch Average Loss: 2.026555
Epoch Average Loss: 2.025246
Epoch Average Loss: 2.023879
Validate Acc: 0.264
Epoch Average Loss: 2.022807
Epoch Average Loss: 2.021365
Epoch Average Loss: 2.020400
Validate Acc: 0.256
Epoch Average Loss: 2.019019
Epoch Average Loss: 2.017685
Epoch Average Loss: 2.016481
Validate Acc: 0.260
Epoch Average Loss: 2.015223
Epoch Average Loss: 2.013901
Epoch Average Loss: 2.012642
Validate Acc: 0.264
Epoch Average Loss: 2.011623
Epoch Average Loss: 2.010306
Epoch Average Loss: 2.008473
Validate Acc: 0.260
Epoch Average Loss: 2.007575
Epoch Average Loss: 2.006137
Epoch Average Loss: 2.004560
```

```
Validate Acc: 0.264
Epoch Average Loss: 2.003235
Epoch Average Loss: 2.001494
Epoch Average Loss: 1.999827
Validate Acc: 0.284
Epoch Average Loss: 1.998114
Epoch Average Loss: 1.996309
Epoch Average Loss: 1.994329
Validate Acc: 0.288
Epoch Average Loss: 1.992309
Epoch Average Loss: 1.990600
Epoch Average Loss: 1.988072
Validate Acc: 0.272
Epoch Average Loss: 1.986238
Epoch Average Loss: 1.984061
Epoch Average Loss: 1.981524
Validate Acc: 0.272
Epoch Average Loss: 1.979365
Epoch Average Loss: 1.977379
Epoch Average Loss: 1.974764
Validate Acc: 0.276
Epoch Average Loss: 1.972478
Epoch Average Loss: 1.970383
Epoch Average Loss: 1.968435
Validate Acc: 0.276
Epoch Average Loss: 1.966031
Epoch Average Loss: 1.964479
Epoch Average Loss: 1.962515
Validate Acc: 0.264
Epoch Average Loss: 1.960328
Epoch Average Loss: 1.958778
Epoch Average Loss: 1.956944
Validate Acc: 0.264
Epoch Average Loss: 1.955720
Epoch Average Loss: 1.954605
Epoch Average Loss: 1.952892
Validate Acc: 0.260
Epoch Average Loss: 1.951665
Epoch Average Loss: 1.950396
Epoch Average Loss: 1.948771
Validate Acc: 0.264
Epoch Average Loss: 1.947941
Epoch Average Loss: 1.946434
Epoch Average Loss: 1.945457
Validate Acc: 0.272
Epoch Average Loss: 1.944042
Epoch Average Loss: 1.943626
Epoch Average Loss: 1.942291
```

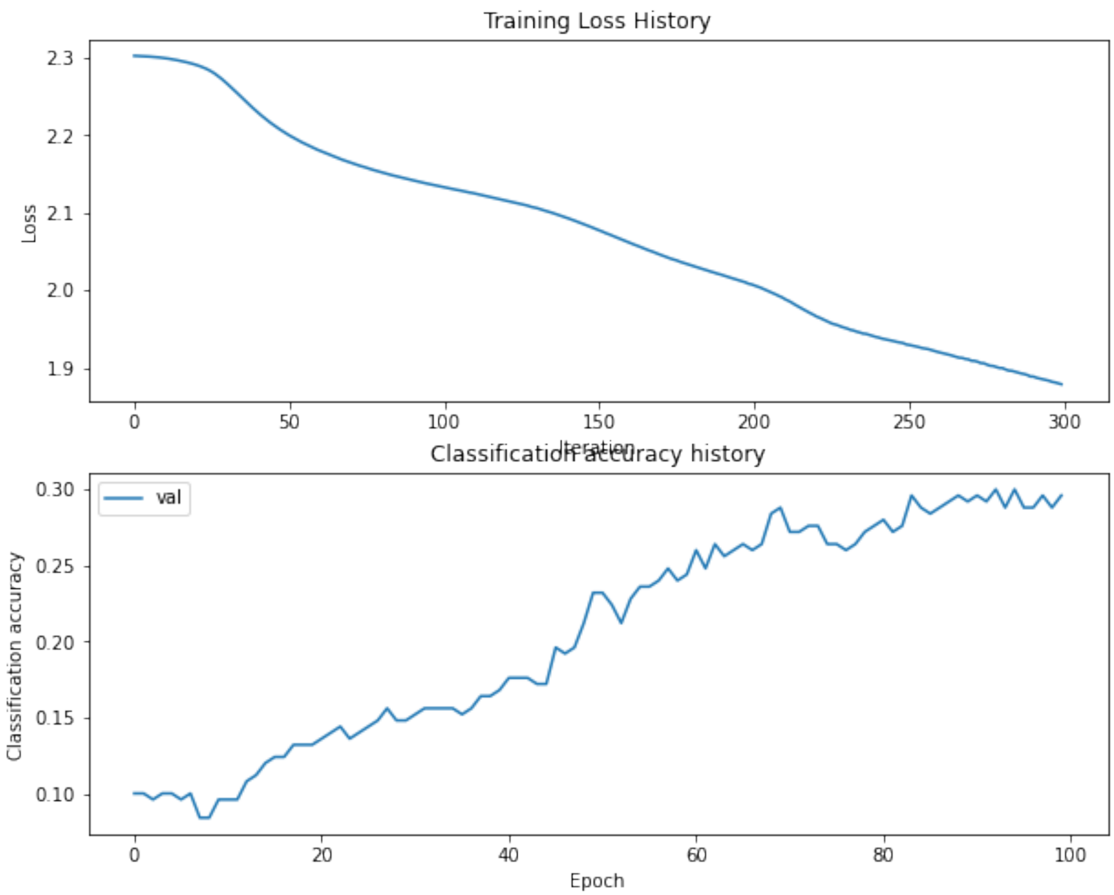
```
Validate Acc: 0.276
Epoch Average Loss: 1.940825
Epoch Average Loss: 1.939869
Epoch Average Loss: 1.938971
Validate Acc: 0.280
Epoch Average Loss: 1.937665
Epoch Average Loss: 1.936849
Epoch Average Loss: 1.936099
Validate Acc: 0.272
Epoch Average Loss: 1.935049
Epoch Average Loss: 1.934202
Epoch Average Loss: 1.933168
Validate Acc: 0.276
Epoch Average Loss: 1.932452
Epoch Average Loss: 1.931721
Epoch Average Loss: 1.929866
Validate Acc: 0.296
Epoch Average Loss: 1.929288
Epoch Average Loss: 1.928296
Epoch Average Loss: 1.927513
Validate Acc: 0.288
Epoch Average Loss: 1.926589
Epoch Average Loss: 1.925377
Epoch Average Loss: 1.924597
Validate Acc: 0.284
Epoch Average Loss: 1.924021
Epoch Average Loss: 1.922756
Epoch Average Loss: 1.921668
Validate Acc: 0.288
Epoch Average Loss: 1.920435
Epoch Average Loss: 1.919496
Epoch Average Loss: 1.918351
Validate Acc: 0.292
Epoch Average Loss: 1.917581
Epoch Average Loss: 1.916469
Epoch Average Loss: 1.915281
Validate Acc: 0.296
Epoch Average Loss: 1.914183
Epoch Average Loss: 1.912858
Epoch Average Loss: 1.912609
Validate Acc: 0.292
Epoch Average Loss: 1.911271
Epoch Average Loss: 1.910731
Epoch Average Loss: 1.908893
Validate Acc: 0.296
Epoch Average Loss: 1.908719
Epoch Average Loss: 1.907627
Epoch Average Loss: 1.905909
```

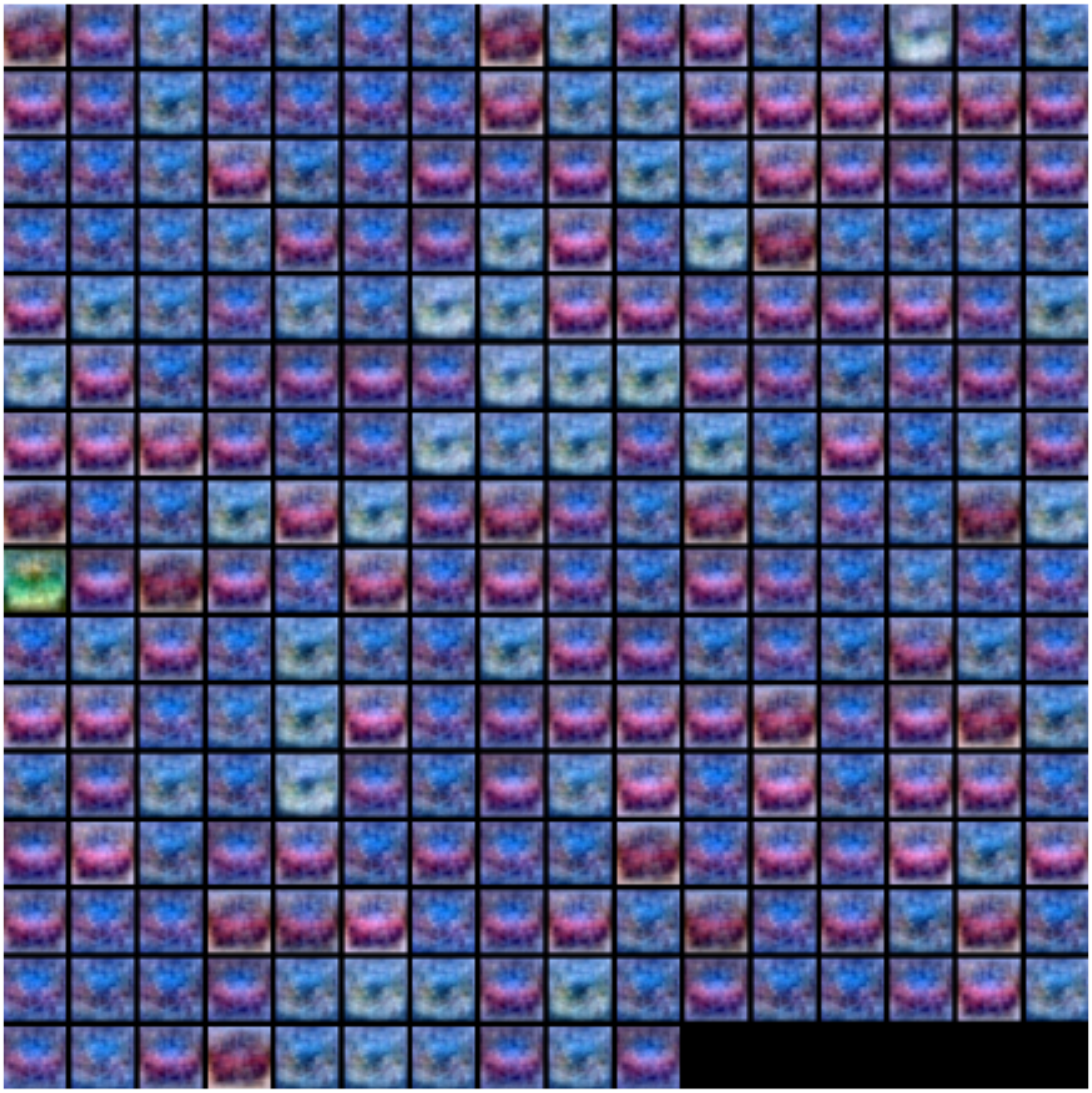
```
Validate Acc: 0.292
Epoch Average Loss: 1.905691
Epoch Average Loss: 1.903903
Epoch Average Loss: 1.902524
Validate Acc: 0.300
Epoch Average Loss: 1.902162
Epoch Average Loss: 1.900993
Epoch Average Loss: 1.899598
Validate Acc: 0.288
Epoch Average Loss: 1.899494
Epoch Average Loss: 1.897645
Epoch Average Loss: 1.896247
Validate Acc: 0.300
Epoch Average Loss: 1.895960
Epoch Average Loss: 1.894918
Epoch Average Loss: 1.893686
Validate Acc: 0.288
Epoch Average Loss: 1.892561
Epoch Average Loss: 1.891756
Epoch Average Loss: 1.890573
Validate Acc: 0.288
Epoch Average Loss: 1.888629
Epoch Average Loss: 1.888515
Epoch Average Loss: 1.886917
Validate Acc: 0.296
Epoch Average Loss: 1.886002
Epoch Average Loss: 1.884938
Epoch Average Loss: 1.884360
Validate Acc: 0.288
Epoch Average Loss: 1.883151
Epoch Average Loss: 1.881703
Epoch Average Loss: 1.880913
Validate Acc: 0.296
Epoch Average Loss: 1.879667
Epoch Average Loss: 1.878622
Training acc:  0.311
Validation acc:  0.3
```

[10]:

```
from ece285.utils.vis_utils import visualize_grid
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()
```

[11]:
```
# TODO: Plot the training_error and validation_accuracy of the best network (5%)
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()

# TODO: visualize the weights of the best network (5%)
show_net_weights(net)
```





[13]:
```
lr = 0.01
weight_decay = 0.001
epoch = 300
hidden_size = 250

dataset = DataLoader(x_train, y_train, x_val, y_val, x_test, y_test)
net = init_model()
optim = SGD(net, lr, weight_decay)
loss_func = CrossEntropyLoss()
batch_size = 200  # (Reduce the batch size if your computer is unable to handle␣
 ↪it)

trainer = Trainer(
```

```python
    dataset, optim, net, loss_func, epoch, batch_size, validate_interval=3
)


train_error, validation_accuracy = trainer.train()

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)
```

```
Epoch Average Loss: 2.302489
Validate Acc: 0.100
Epoch Average Loss: 2.302175
Epoch Average Loss: 2.301808
Epoch Average Loss: 2.301294
Validate Acc: 0.100
Epoch Average Loss: 2.300603
Epoch Average Loss: 2.299667
Epoch Average Loss: 2.298424
Validate Acc: 0.096
Epoch Average Loss: 2.296938
Epoch Average Loss: 2.295239
Epoch Average Loss: 2.293194
Validate Acc: 0.100
Epoch Average Loss: 2.290830
Epoch Average Loss: 2.287830
Epoch Average Loss: 2.284016
Validate Acc: 0.088
Epoch Average Loss: 2.279108
Epoch Average Loss: 2.272916
Epoch Average Loss: 2.265835
Validate Acc: 0.096
Epoch Average Loss: 2.258275
Epoch Average Loss: 2.250465
Epoch Average Loss: 2.242635
Validate Acc: 0.108
Epoch Average Loss: 2.234885
Epoch Average Loss: 2.227745
Epoch Average Loss: 2.220971
Validate Acc: 0.124
Epoch Average Loss: 2.214528
Epoch Average Loss: 2.208796
Epoch Average Loss: 2.203476
Validate Acc: 0.128
```

```
Epoch Average Loss: 2.198347
Epoch Average Loss: 2.193775
Epoch Average Loss: 2.189463
Validate Acc: 0.132
Epoch Average Loss: 2.185296
Epoch Average Loss: 2.181654
Epoch Average Loss: 2.177800
Validate Acc: 0.140
Epoch Average Loss: 2.174524
Epoch Average Loss: 2.171149
Epoch Average Loss: 2.168381
Validate Acc: 0.144
Epoch Average Loss: 2.165116
Epoch Average Loss: 2.162130
Epoch Average Loss: 2.159715
Validate Acc: 0.140
Epoch Average Loss: 2.157130
Epoch Average Loss: 2.154534
Epoch Average Loss: 2.151824
Validate Acc: 0.148
Epoch Average Loss: 2.149558
Epoch Average Loss: 2.147127
Epoch Average Loss: 2.145511
Validate Acc: 0.152
Epoch Average Loss: 2.142997
Epoch Average Loss: 2.140888
Epoch Average Loss: 2.138844
Validate Acc: 0.148
Epoch Average Loss: 2.137320
Epoch Average Loss: 2.135263
Epoch Average Loss: 2.133492
Validate Acc: 0.156
Epoch Average Loss: 2.131207
Epoch Average Loss: 2.130287
Epoch Average Loss: 2.128015
Validate Acc: 0.164
Epoch Average Loss: 2.126481
Epoch Average Loss: 2.124808
Epoch Average Loss: 2.122231
Validate Acc: 0.144
Epoch Average Loss: 2.121117
Epoch Average Loss: 2.119410
Epoch Average Loss: 2.117811
Validate Acc: 0.156
Epoch Average Loss: 2.115939
Epoch Average Loss: 2.114163
Epoch Average Loss: 2.111996
Validate Acc: 0.168
```

```
Epoch Average Loss: 2.110252
Epoch Average Loss: 2.107744
Epoch Average Loss: 2.105854
Validate Acc: 0.164
Epoch Average Loss: 2.103644
Epoch Average Loss: 2.101105
Epoch Average Loss: 2.098240
Validate Acc: 0.180
Epoch Average Loss: 2.095810
Epoch Average Loss: 2.092167
Epoch Average Loss: 2.090272
Validate Acc: 0.192
Epoch Average Loss: 2.087279
Epoch Average Loss: 2.083990
Epoch Average Loss: 2.080885
Validate Acc: 0.228
Epoch Average Loss: 2.077561
Epoch Average Loss: 2.074329
Epoch Average Loss: 2.070946
Validate Acc: 0.232
Epoch Average Loss: 2.067553
Epoch Average Loss: 2.063935
Epoch Average Loss: 2.060616
Validate Acc: 0.228
Epoch Average Loss: 2.057186
Epoch Average Loss: 2.053586
Epoch Average Loss: 2.050185
Validate Acc: 0.236
Epoch Average Loss: 2.047432
Epoch Average Loss: 2.044399
Epoch Average Loss: 2.041291
Validate Acc: 0.240
Epoch Average Loss: 2.038023
Epoch Average Loss: 2.034906
Epoch Average Loss: 2.031714
Validate Acc: 0.252
Epoch Average Loss: 2.029569
Epoch Average Loss: 2.026602
Epoch Average Loss: 2.023012
Validate Acc: 0.268
Epoch Average Loss: 2.021186
Epoch Average Loss: 2.019157
Epoch Average Loss: 2.015975
Validate Acc: 0.256
Epoch Average Loss: 2.014066
Epoch Average Loss: 2.011505
Epoch Average Loss: 2.008893
Validate Acc: 0.276
```

```
Epoch Average Loss: 2.006200
Epoch Average Loss: 2.004285
Epoch Average Loss: 2.002319
Validate Acc: 0.276
Epoch Average Loss: 2.000018
Epoch Average Loss: 1.997167
Epoch Average Loss: 1.994596
Validate Acc: 0.272
Epoch Average Loss: 1.992388
Epoch Average Loss: 1.989815
Epoch Average Loss: 1.987662
Validate Acc: 0.280
Epoch Average Loss: 1.984376
Epoch Average Loss: 1.982268
Epoch Average Loss: 1.978841
Validate Acc: 0.280
Epoch Average Loss: 1.976054
Epoch Average Loss: 1.973328
Epoch Average Loss: 1.969381
Validate Acc: 0.288
Epoch Average Loss: 1.964949
Epoch Average Loss: 1.961530
Epoch Average Loss: 1.958092
Validate Acc: 0.268
Epoch Average Loss: 1.954897
Epoch Average Loss: 1.949979
Epoch Average Loss: 1.947427
Validate Acc: 0.276
Epoch Average Loss: 1.943943
Epoch Average Loss: 1.941818
Epoch Average Loss: 1.938477
Validate Acc: 0.268
Epoch Average Loss: 1.936221
Epoch Average Loss: 1.933333
Epoch Average Loss: 1.930949
Validate Acc: 0.272
Epoch Average Loss: 1.928596
Epoch Average Loss: 1.926733
Epoch Average Loss: 1.923987
Validate Acc: 0.276
Epoch Average Loss: 1.921392
Epoch Average Loss: 1.920596
Epoch Average Loss: 1.916185
Validate Acc: 0.292
Epoch Average Loss: 1.915205
Epoch Average Loss: 1.913602
Epoch Average Loss: 1.910021
Validate Acc: 0.276
```

```
Epoch Average Loss: 1.907992
Epoch Average Loss: 1.906756
Epoch Average Loss: 1.904837
Validate Acc: 0.296
Epoch Average Loss: 1.900463
Epoch Average Loss: 1.899878
Epoch Average Loss: 1.897696
Validate Acc: 0.288
Epoch Average Loss: 1.894868
Epoch Average Loss: 1.892308
Epoch Average Loss: 1.890190
Validate Acc: 0.280
Epoch Average Loss: 1.888111
Epoch Average Loss: 1.885584
Epoch Average Loss: 1.882782
Validate Acc: 0.280
Epoch Average Loss: 1.880600
Epoch Average Loss: 1.880736
Epoch Average Loss: 1.878126
Validate Acc: 0.312
Epoch Average Loss: 1.873254
Epoch Average Loss: 1.871680
Epoch Average Loss: 1.870764
Validate Acc: 0.288
Epoch Average Loss: 1.867330
Epoch Average Loss: 1.865461
Epoch Average Loss: 1.863002
Validate Acc: 0.296
Epoch Average Loss: 1.861743
Epoch Average Loss: 1.859313
Epoch Average Loss: 1.857925
Validate Acc: 0.316
Epoch Average Loss: 1.855220
Epoch Average Loss: 1.852505
Epoch Average Loss: 1.850832
Validate Acc: 0.300
Epoch Average Loss: 1.849007
Epoch Average Loss: 1.846634
Epoch Average Loss: 1.843804
Validate Acc: 0.308
Epoch Average Loss: 1.842197
Validate Acc: 0.296
Epoch Average Loss: 1.835131
Epoch Average Loss: 1.832353
Epoch Average Loss: 1.829254
Validate Acc: 0.288
Epoch Average Loss: 1.827850
Epoch Average Loss: 1.824839
```

```
Epoch Average Loss: 1.823416
Validate Acc: 0.312
Epoch Average Loss: 1.821212
Epoch Average Loss: 1.818873
Epoch Average Loss: 1.817688
Validate Acc: 0.312
Epoch Average Loss: 1.813504
Epoch Average Loss: 1.813492
Epoch Average Loss: 1.810200
Validate Acc: 0.316
Epoch Average Loss: 1.808298
Epoch Average Loss: 1.808731
Epoch Average Loss: 1.803758
Validate Acc: 0.300
Epoch Average Loss: 1.802092
Epoch Average Loss: 1.800509
Epoch Average Loss: 1.798266
Validate Acc: 0.316
Epoch Average Loss: 1.794417
Epoch Average Loss: 1.792550
Epoch Average Loss: 1.789978
Validate Acc: 0.332
Epoch Average Loss: 1.789146
Epoch Average Loss: 1.787217
Epoch Average Loss: 1.784517
Validate Acc: 0.320
Epoch Average Loss: 1.781417
Epoch Average Loss: 1.778753
Epoch Average Loss: 1.776784
Validate Acc: 0.344
Epoch Average Loss: 1.774150
Epoch Average Loss: 1.773725
Epoch Average Loss: 1.774651
Validate Acc: 0.320
Epoch Average Loss: 1.773029
Epoch Average Loss: 1.768910
Epoch Average Loss: 1.766151
Validate Acc: 0.324
Epoch Average Loss: 1.764716
Epoch Average Loss: 1.762030
Epoch Average Loss: 1.760418
Validate Acc: 0.328
Epoch Average Loss: 1.755896
Epoch Average Loss: 1.757062
Epoch Average Loss: 1.754094
Validate Acc: 0.360
Epoch Average Loss: 1.752389
Epoch Average Loss: 1.750901
```

```
Epoch Average Loss: 1.747780
Validate Acc: 0.336
Epoch Average Loss: 1.746178
Epoch Average Loss: 1.743281
Epoch Average Loss: 1.742734
Validate Acc: 0.348
Epoch Average Loss: 1.739866
Epoch Average Loss: 1.737988
Epoch Average Loss: 1.738288
Validate Acc: 0.360
Epoch Average Loss: 1.735300
Epoch Average Loss: 1.731379
Epoch Average Loss: 1.733487
Validate Acc: 0.360
Epoch Average Loss: 1.729472
Epoch Average Loss: 1.727046
Epoch Average Loss: 1.725056
Validate Acc: 0.360
Epoch Average Loss: 1.723247
Epoch Average Loss: 1.722433
Epoch Average Loss: 1.721847
Validate Acc: 0.368
Epoch Average Loss: 1.717599
Epoch Average Loss: 1.717074
Epoch Average Loss: 1.716074
Validate Acc: 0.384
Epoch Average Loss: 1.711912
Epoch Average Loss: 1.710651
Epoch Average Loss: 1.707093
Validate Acc: 0.364
Epoch Average Loss: 1.707324
Epoch Average Loss: 1.705904
Epoch Average Loss: 1.704887
Validate Acc: 0.372
Epoch Average Loss: 1.704297
Epoch Average Loss: 1.701699
Epoch Average Loss: 1.698358
Validate Acc: 0.364
Epoch Average Loss: 1.697639
Epoch Average Loss: 1.698685
Epoch Average Loss: 1.692287
Validate Acc: 0.372
Epoch Average Loss: 1.692875
Epoch Average Loss: 1.688885
Epoch Average Loss: 1.688290
Validate Acc: 0.380
Epoch Average Loss: 1.686823
Epoch Average Loss: 1.686121
```

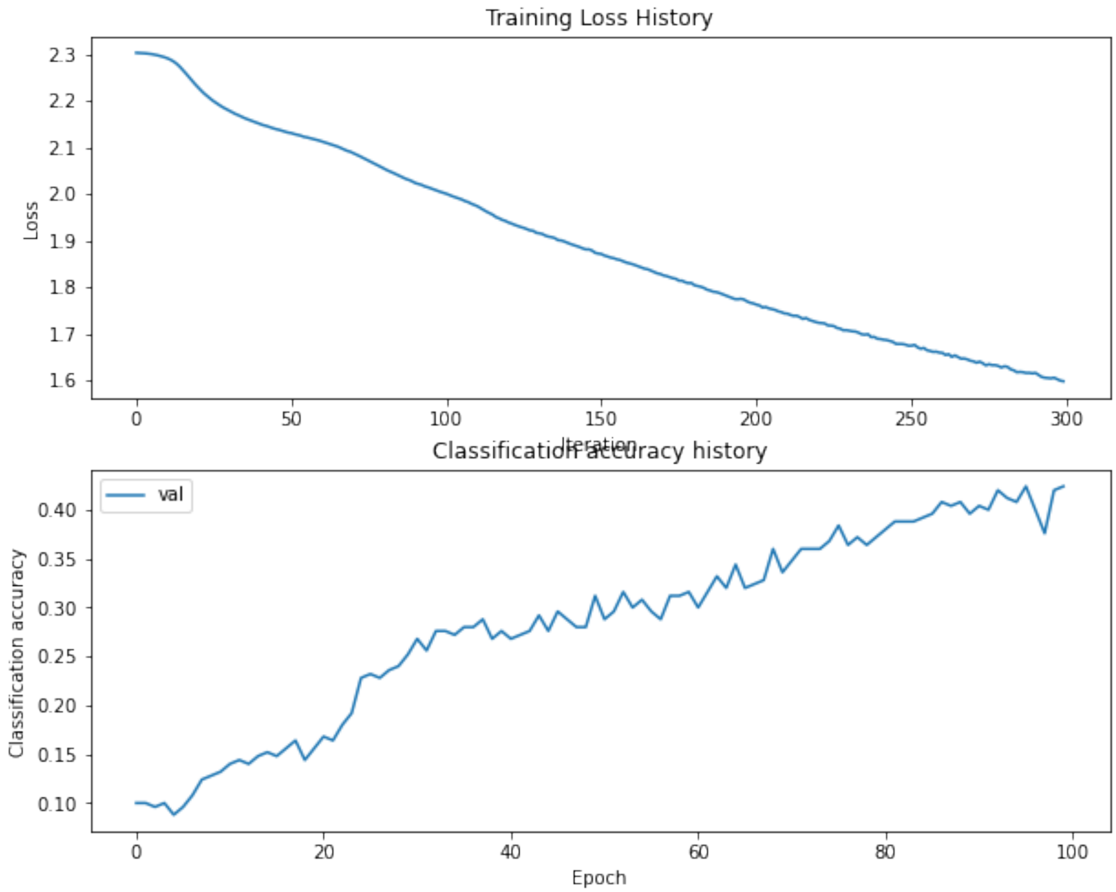
```
Epoch Average Loss: 1.684055
Validate Acc: 0.388
Epoch Average Loss: 1.682240
Epoch Average Loss: 1.677842
Epoch Average Loss: 1.677588
Validate Acc: 0.388
Epoch Average Loss: 1.677669
Epoch Average Loss: 1.676395
Epoch Average Loss: 1.673974
Validate Acc: 0.388
Epoch Average Loss: 1.674011
Epoch Average Loss: 1.675739
Epoch Average Loss: 1.670373
Validate Acc: 0.392
Epoch Average Loss: 1.667458
Epoch Average Loss: 1.669270
Epoch Average Loss: 1.664216
Validate Acc: 0.396
Epoch Average Loss: 1.662761
Epoch Average Loss: 1.660724
Epoch Average Loss: 1.661186
Validate Acc: 0.408
Epoch Average Loss: 1.659134
Epoch Average Loss: 1.658604
Epoch Average Loss: 1.653877
Validate Acc: 0.404
Epoch Average Loss: 1.655886
Epoch Average Loss: 1.650004
Epoch Average Loss: 1.653319
Validate Acc: 0.408
Epoch Average Loss: 1.649841
Epoch Average Loss: 1.646097
Epoch Average Loss: 1.646471
Validate Acc: 0.396
Epoch Average Loss: 1.644635
Epoch Average Loss: 1.641709
Epoch Average Loss: 1.640410
Validate Acc: 0.404
Epoch Average Loss: 1.637459
Epoch Average Loss: 1.639823
Epoch Average Loss: 1.636308
Validate Acc: 0.400
Epoch Average Loss: 1.631699
Epoch Average Loss: 1.634376
Epoch Average Loss: 1.632244
Validate Acc: 0.420
Epoch Average Loss: 1.632382
Epoch Average Loss: 1.630535
```

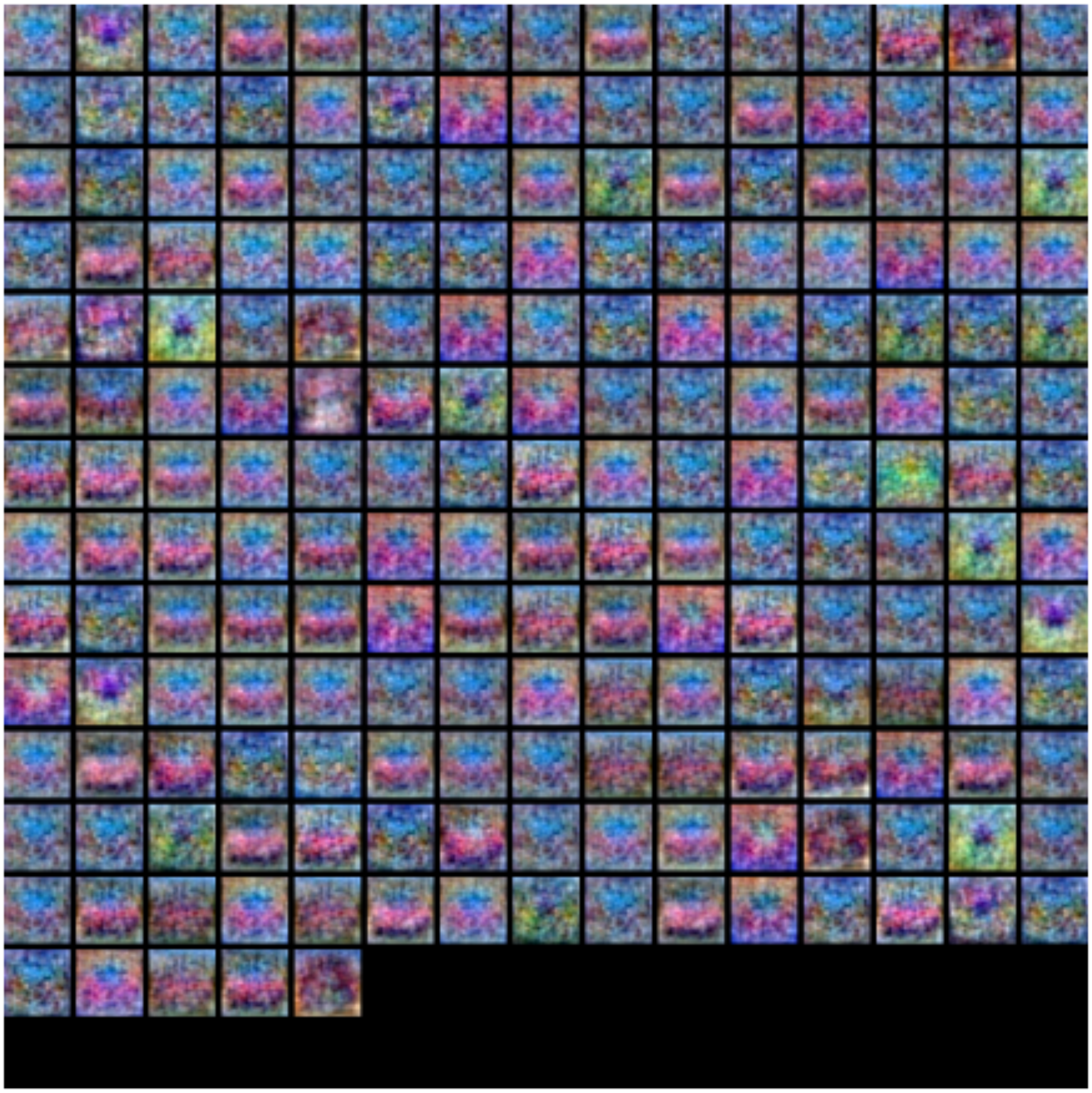
```
Epoch Average Loss: 1.627151
Validate Acc: 0.412
Epoch Average Loss: 1.629897
Epoch Average Loss: 1.628444
Epoch Average Loss: 1.623241
Validate Acc: 0.408
Epoch Average Loss: 1.621192
Epoch Average Loss: 1.617033
Epoch Average Loss: 1.617744
Validate Acc: 0.424
Epoch Average Loss: 1.617058
Epoch Average Loss: 1.615454
Epoch Average Loss: 1.615704
Validate Acc: 0.400
Epoch Average Loss: 1.614581
Epoch Average Loss: 1.615586
Epoch Average Loss: 1.611197
Validate Acc: 0.376
Epoch Average Loss: 1.606902
Epoch Average Loss: 1.605216
Epoch Average Loss: 1.604763
Validate Acc: 0.420
Epoch Average Loss: 1.603948
Epoch Average Loss: 1.605291
Epoch Average Loss: 1.602218
Validate Acc: 0.424
Epoch Average Loss: 1.599011
Epoch Average Loss: 1.597734
Training acc:  0.4318
Validation acc:  0.428
```

[14]:

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

```
# TODO: visualize the weights of the best network (5%)
show_net_weights(net)
```





[12]:
```
lr = 0.01
weight_decay = 0.005
epoch = 300
hidden_size = 220


dataset = DataLoader(x_train, y_train, x_val, y_val, x_test, y_test)
best_net = init_model()
optim = SGD(best_net, lr, weight_decay)
loss_func = CrossEntropyLoss()
batch_size = 200  # (Reduce the batch size if your computer is unable to handle␣
↪it)
```

```
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("Training acc: ", acc)
out_val = best_net.predict(x_val)
acc = get_classification_accuracy(out_val, y_val)
print("Validation acc: ", acc)
```

```
Epoch Average Loss: 2.302472
Validate Acc: 0.088
Epoch Average Loss: 2.302144
Epoch Average Loss: 2.301731
Epoch Average Loss: 2.301198
Validate Acc: 0.096
Epoch Average Loss: 2.300464
Epoch Average Loss: 2.299494
Epoch Average Loss: 2.298254
Validate Acc: 0.092
Epoch Average Loss: 2.296750
Epoch Average Loss: 2.294986
Epoch Average Loss: 2.292963
Validate Acc: 0.084
Epoch Average Loss: 2.290522
Epoch Average Loss: 2.287571
Epoch Average Loss: 2.283761
Validate Acc: 0.088
Epoch Average Loss: 2.278771
Epoch Average Loss: 2.272477
Epoch Average Loss: 2.265415
Validate Acc: 0.096
Epoch Average Loss: 2.257865
Epoch Average Loss: 2.250176
Epoch Average Loss: 2.242396
Validate Acc: 0.108
Epoch Average Loss: 2.234918
Epoch Average Loss: 2.227660
Epoch Average Loss: 2.221030
Validate Acc: 0.120
Epoch Average Loss: 2.214699
Epoch Average Loss: 2.209097
Epoch Average Loss: 2.203624
```

```
Validate Acc: 0.124
Epoch Average Loss: 2.199013
Epoch Average Loss: 2.194167
Epoch Average Loss: 2.189933
Validate Acc: 0.132
Epoch Average Loss: 2.185788
Epoch Average Loss: 2.182165
Epoch Average Loss: 2.178585
Validate Acc: 0.136
Epoch Average Loss: 2.175092
Epoch Average Loss: 2.171694
Epoch Average Loss: 2.168471
Validate Acc: 0.136
Epoch Average Loss: 2.165718
Epoch Average Loss: 2.163014
Epoch Average Loss: 2.160226
Validate Acc: 0.140
Epoch Average Loss: 2.157685
Epoch Average Loss: 2.155111
Epoch Average Loss: 2.152674
Validate Acc: 0.140
Epoch Average Loss: 2.150621
Epoch Average Loss: 2.148555
Epoch Average Loss: 2.146032
Validate Acc: 0.144
Epoch Average Loss: 2.144091
Epoch Average Loss: 2.141980
Epoch Average Loss: 2.140193
Validate Acc: 0.160
Epoch Average Loss: 2.138349
Epoch Average Loss: 2.136543
Epoch Average Loss: 2.134345
Validate Acc: 0.148
Epoch Average Loss: 2.132810
Epoch Average Loss: 2.131095
Epoch Average Loss: 2.129480
Validate Acc: 0.160
Epoch Average Loss: 2.127636
Epoch Average Loss: 2.126270
Epoch Average Loss: 2.124454
Validate Acc: 0.164
Epoch Average Loss: 2.122775
Epoch Average Loss: 2.121050
Epoch Average Loss: 2.119146
Validate Acc: 0.172
Epoch Average Loss: 2.117335
Epoch Average Loss: 2.115456
Epoch Average Loss: 2.113508
```

```
Validate Acc: 0.172
Epoch Average Loss: 2.111606
Epoch Average Loss: 2.109921
Epoch Average Loss: 2.107675
Validate Acc: 0.172
Epoch Average Loss: 2.105146
Epoch Average Loss: 2.102636
Epoch Average Loss: 2.100345
Validate Acc: 0.172
Epoch Average Loss: 2.097641
Epoch Average Loss: 2.095070
Epoch Average Loss: 2.092407
Validate Acc: 0.180
Epoch Average Loss: 2.089245
Epoch Average Loss: 2.085959
Epoch Average Loss: 2.082830
Validate Acc: 0.220
Epoch Average Loss: 2.079610
Epoch Average Loss: 2.076454
Epoch Average Loss: 2.073281
Validate Acc: 0.216
Epoch Average Loss: 2.069660
Epoch Average Loss: 2.066625
Epoch Average Loss: 2.062764
Validate Acc: 0.220
Epoch Average Loss: 2.059428
Epoch Average Loss: 2.056640
Epoch Average Loss: 2.053280
Validate Acc: 0.232
Epoch Average Loss: 2.049563
Epoch Average Loss: 2.046491
Epoch Average Loss: 2.043952
Validate Acc: 0.240
Epoch Average Loss: 2.040936
Epoch Average Loss: 2.037675
Epoch Average Loss: 2.035606
Validate Acc: 0.260
Epoch Average Loss: 2.032410
Epoch Average Loss: 2.030140
Epoch Average Loss: 2.027801
Validate Acc: 0.252
Epoch Average Loss: 2.024898
Epoch Average Loss: 2.022218
Epoch Average Loss: 2.019796
Validate Acc: 0.244
Epoch Average Loss: 2.017924
Epoch Average Loss: 2.015195
Epoch Average Loss: 2.012998
```

```
Validate Acc: 0.260
Epoch Average Loss: 2.011441
Epoch Average Loss: 2.008841
Epoch Average Loss: 2.006403
Validate Acc: 0.256
Epoch Average Loss: 2.004663
Epoch Average Loss: 2.002702
Epoch Average Loss: 2.000946
Validate Acc: 0.272
Epoch Average Loss: 1.998283
Epoch Average Loss: 1.996196
Epoch Average Loss: 1.994684
Validate Acc: 0.276
Epoch Average Loss: 1.992475
Epoch Average Loss: 1.990613
Epoch Average Loss: 1.988349
Validate Acc: 0.276
Epoch Average Loss: 1.986744
Epoch Average Loss: 1.984256
Epoch Average Loss: 1.981858
Validate Acc: 0.276
Epoch Average Loss: 1.980328
Epoch Average Loss: 1.979070
Epoch Average Loss: 1.975789
Validate Acc: 0.292
Epoch Average Loss: 1.973444
Epoch Average Loss: 1.970607
Epoch Average Loss: 1.967387
Validate Acc: 0.288
Epoch Average Loss: 1.964749
Epoch Average Loss: 1.961034
Epoch Average Loss: 1.957519
Validate Acc: 0.296
Epoch Average Loss: 1.954154
Epoch Average Loss: 1.950538
Epoch Average Loss: 1.947467
Validate Acc: 0.284
Epoch Average Loss: 1.944108
Epoch Average Loss: 1.940550
Epoch Average Loss: 1.937393
Validate Acc: 0.264
Epoch Average Loss: 1.935336
Epoch Average Loss: 1.932406
Epoch Average Loss: 1.928246
Validate Acc: 0.284
Epoch Average Loss: 1.927339
Epoch Average Loss: 1.923892
Epoch Average Loss: 1.921886
```

```
Validate Acc: 0.292
Epoch Average Loss: 1.917909
Epoch Average Loss: 1.916347
Epoch Average Loss: 1.913984
Validate Acc: 0.284
Epoch Average Loss: 1.910486
Epoch Average Loss: 1.910019
Epoch Average Loss: 1.907236
Validate Acc: 0.288
Epoch Average Loss: 1.906062
Epoch Average Loss: 1.903019
Epoch Average Loss: 1.901872
Validate Acc: 0.292
Epoch Average Loss: 1.899603
Epoch Average Loss: 1.896029
Epoch Average Loss: 1.893644
Validate Acc: 0.288
Epoch Average Loss: 1.891181
Epoch Average Loss: 1.889215
Epoch Average Loss: 1.888081
Validate Acc: 0.308
Epoch Average Loss: 1.884229
Epoch Average Loss: 1.882331
Epoch Average Loss: 1.879765
Validate Acc: 0.312
Epoch Average Loss: 1.878998
Epoch Average Loss: 1.875771
Epoch Average Loss: 1.874489
Validate Acc: 0.288
Epoch Average Loss: 1.872152
Epoch Average Loss: 1.870157
Epoch Average Loss: 1.868175
Validate Acc: 0.304
Epoch Average Loss: 1.865281
Epoch Average Loss: 1.862937
Epoch Average Loss: 1.860678
Validate Acc: 0.296
Epoch Average Loss: 1.857830
Epoch Average Loss: 1.856804
Epoch Average Loss: 1.853794
Validate Acc: 0.288
Epoch Average Loss: 1.852897
Epoch Average Loss: 1.849328
Epoch Average Loss: 1.847137
Validate Acc: 0.324
Epoch Average Loss: 1.847143
Epoch Average Loss: 1.842732
Epoch Average Loss: 1.840457
```

```
Validate Acc: 0.316
Epoch Average Loss: 1.838717
Epoch Average Loss: 1.837108
Epoch Average Loss: 1.834021
Validate Acc: 0.316
Epoch Average Loss: 1.832217
Epoch Average Loss: 1.830930
Epoch Average Loss: 1.828427
Validate Acc: 0.304
Epoch Average Loss: 1.826136
Epoch Average Loss: 1.822719
Epoch Average Loss: 1.821312
Validate Acc: 0.324
Epoch Average Loss: 1.818760
Epoch Average Loss: 1.815600
Epoch Average Loss: 1.814089
Validate Acc: 0.312
Epoch Average Loss: 1.811374
Epoch Average Loss: 1.808949
Epoch Average Loss: 1.806689
Validate Acc: 0.332
Epoch Average Loss: 1.806491
Epoch Average Loss: 1.802886
Epoch Average Loss: 1.803246
Validate Acc: 0.308
Epoch Average Loss: 1.797665
Epoch Average Loss: 1.796849
Epoch Average Loss: 1.795012
Validate Acc: 0.320
Epoch Average Loss: 1.792252
Epoch Average Loss: 1.790518
Epoch Average Loss: 1.789773
Validate Acc: 0.328
Epoch Average Loss: 1.786349
Epoch Average Loss: 1.783976
Epoch Average Loss: 1.783662
Validate Acc: 0.336
Epoch Average Loss: 1.778963
Epoch Average Loss: 1.778777
Epoch Average Loss: 1.776502
Validate Acc: 0.328
Epoch Average Loss: 1.773648
Epoch Average Loss: 1.771008
Epoch Average Loss: 1.769815
Validate Acc: 0.324
Epoch Average Loss: 1.768022
Epoch Average Loss: 1.765407
Epoch Average Loss: 1.764688
```

```
Validate Acc: 0.344
Epoch Average Loss: 1.761039
Epoch Average Loss: 1.761452
Epoch Average Loss: 1.758851
Validate Acc: 0.332
Epoch Average Loss: 1.755237
Epoch Average Loss: 1.754111
Epoch Average Loss: 1.752083
Validate Acc: 0.356
Epoch Average Loss: 1.751765
Epoch Average Loss: 1.746875
Epoch Average Loss: 1.747984
Validate Acc: 0.360
Epoch Average Loss: 1.743258
Epoch Average Loss: 1.744117
Epoch Average Loss: 1.742469
Validate Acc: 0.372
Epoch Average Loss: 1.740599
Epoch Average Loss: 1.736201
Epoch Average Loss: 1.735823
Validate Acc: 0.360
Epoch Average Loss: 1.733081
Epoch Average Loss: 1.731261
Epoch Average Loss: 1.730866
Validate Acc: 0.360
Epoch Average Loss: 1.730214
Epoch Average Loss: 1.724968
Epoch Average Loss: 1.725022
Validate Acc: 0.356
Epoch Average Loss: 1.723810
Epoch Average Loss: 1.721130
Epoch Average Loss: 1.719231
Validate Acc: 0.368
Epoch Average Loss: 1.716712
Epoch Average Loss: 1.720263
Epoch Average Loss: 1.714299
Validate Acc: 0.360
Epoch Average Loss: 1.712697
Epoch Average Loss: 1.711832
Epoch Average Loss: 1.711066
Validate Acc: 0.368
Epoch Average Loss: 1.706354
Epoch Average Loss: 1.706054
Epoch Average Loss: 1.706269
Validate Acc: 0.364
Epoch Average Loss: 1.701834
Epoch Average Loss: 1.701507
Epoch Average Loss: 1.702851
```

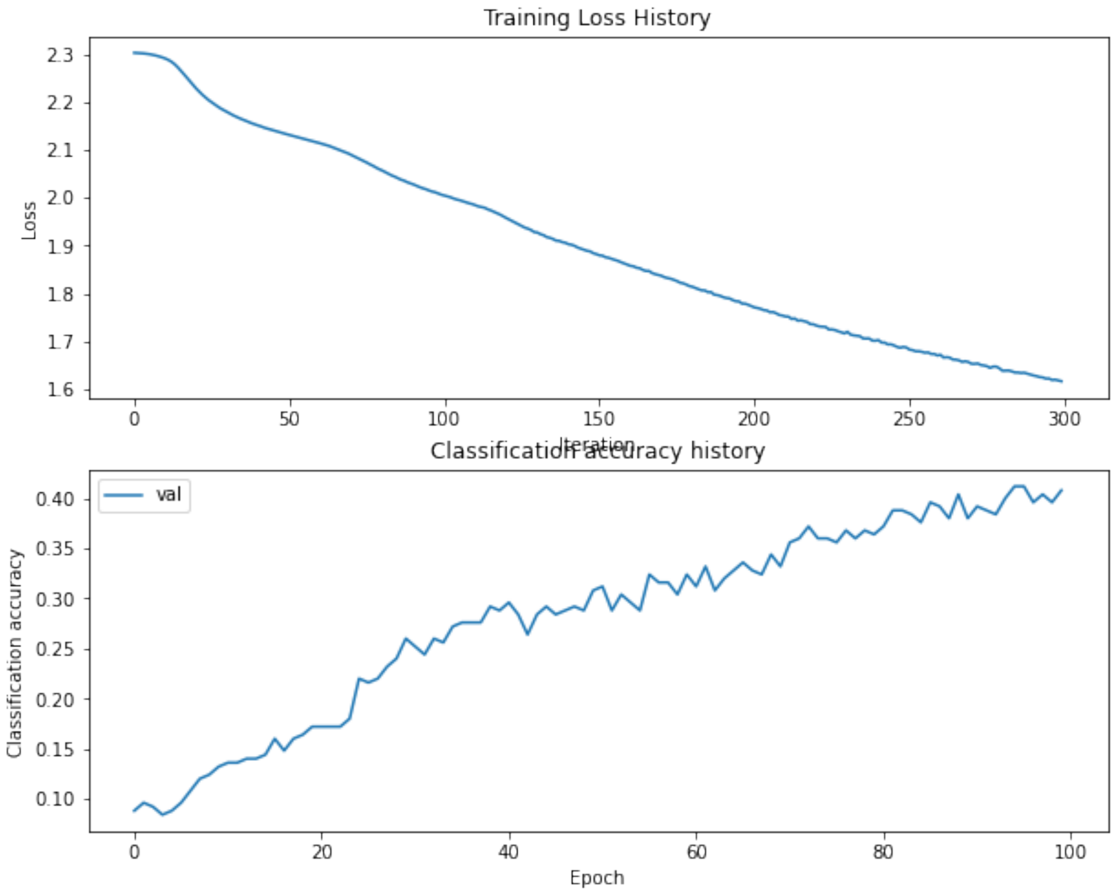
```
Validate Acc: 0.372
Epoch Average Loss: 1.697556
Epoch Average Loss: 1.697616
Epoch Average Loss: 1.693791
Validate Acc: 0.388
Epoch Average Loss: 1.694119
Epoch Average Loss: 1.691758
Epoch Average Loss: 1.688598
Validate Acc: 0.388
Epoch Average Loss: 1.686839
Epoch Average Loss: 1.688522
Epoch Average Loss: 1.687408
Validate Acc: 0.384
Epoch Average Loss: 1.683037
Epoch Average Loss: 1.681893
Epoch Average Loss: 1.679489
Validate Acc: 0.376
Epoch Average Loss: 1.679530
Epoch Average Loss: 1.678412
Epoch Average Loss: 1.676471
Validate Acc: 0.396
Epoch Average Loss: 1.677150
Epoch Average Loss: 1.673772
Epoch Average Loss: 1.673934
Validate Acc: 0.392
Epoch Average Loss: 1.670671
Epoch Average Loss: 1.672209
Epoch Average Loss: 1.666345
Validate Acc: 0.380
Epoch Average Loss: 1.667167
Epoch Average Loss: 1.666193
Epoch Average Loss: 1.662227
Validate Acc: 0.404
Epoch Average Loss: 1.661961
Epoch Average Loss: 1.660465
Epoch Average Loss: 1.657293
Validate Acc: 0.380
Epoch Average Loss: 1.658638
Epoch Average Loss: 1.657067
Epoch Average Loss: 1.653397
Validate Acc: 0.392
Epoch Average Loss: 1.653204
Epoch Average Loss: 1.654069
Epoch Average Loss: 1.650875
Validate Acc: 0.388
Epoch Average Loss: 1.649861
Epoch Average Loss: 1.648341
Epoch Average Loss: 1.644715
```

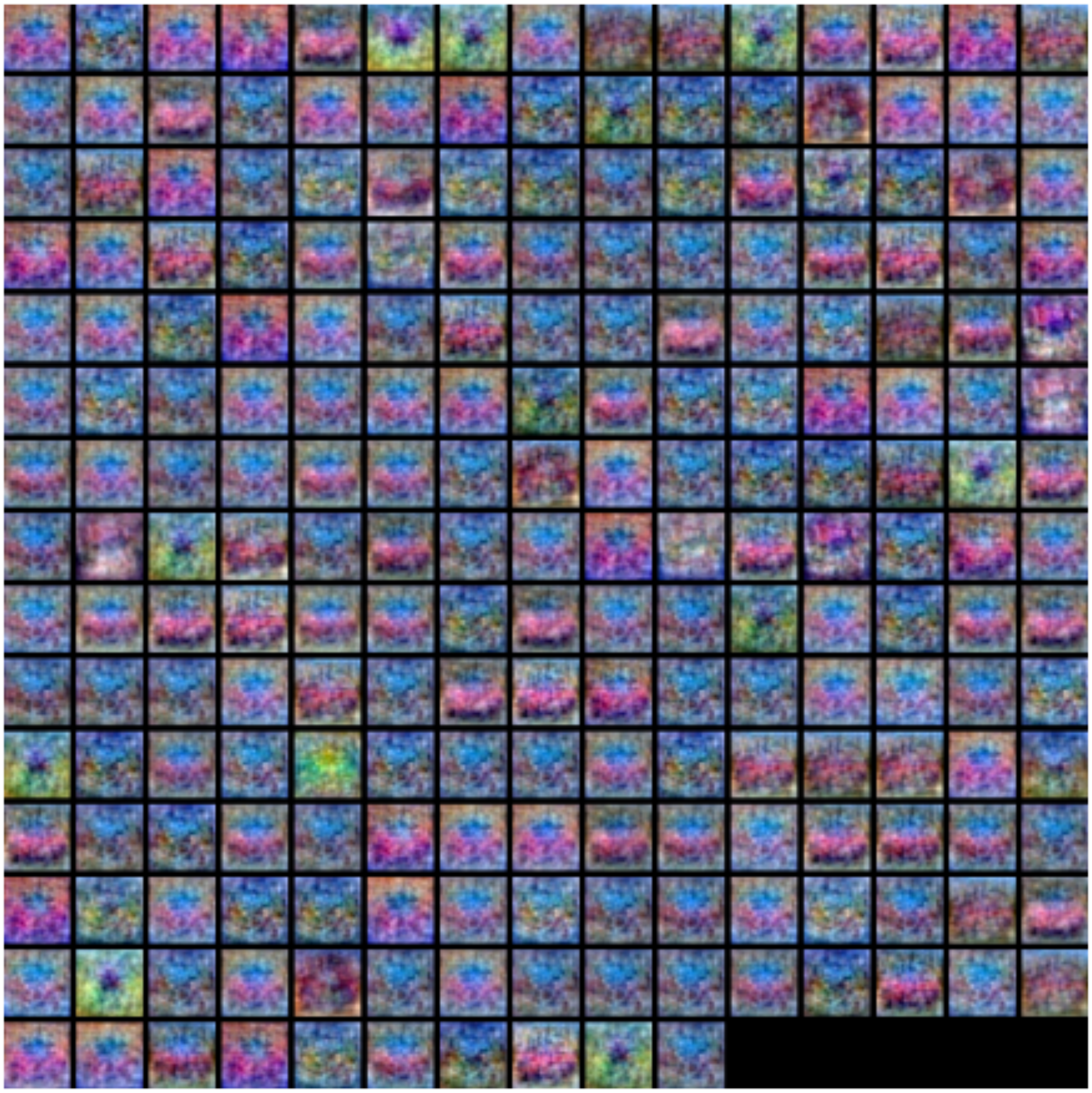
```
Validate Acc: 0.384
Epoch Average Loss: 1.647571
Epoch Average Loss: 1.647479
Epoch Average Loss: 1.643969
Validate Acc: 0.400
Epoch Average Loss: 1.638793
Epoch Average Loss: 1.638986
Epoch Average Loss: 1.639337
Validate Acc: 0.412
Epoch Average Loss: 1.637587
Epoch Average Loss: 1.635226
Epoch Average Loss: 1.635178
Validate Acc: 0.412
Epoch Average Loss: 1.634137
Epoch Average Loss: 1.634714
Epoch Average Loss: 1.632701
Validate Acc: 0.396
Epoch Average Loss: 1.630597
Epoch Average Loss: 1.628989
Epoch Average Loss: 1.627605
Validate Acc: 0.404
Epoch Average Loss: 1.625635
Epoch Average Loss: 1.624906
Epoch Average Loss: 1.622428
Validate Acc: 0.396
Epoch Average Loss: 1.622633
Epoch Average Loss: 1.619554
Epoch Average Loss: 1.620196
Validate Acc: 0.408
Epoch Average Loss: 1.618700
Epoch Average Loss: 1.616955
Training acc:  0.427
Validation acc:  0.432
```

[13]:

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

# TODO: visualize the weights of the best network (5%)
show_net_weights(best_net)
```





```
[14]: acc = get_classification_accuracy(out_train, y_train)
print("Training acc: ", acc)
out_val = best_net.predict(x_val)
acc = get_classification_accuracy(out_val, y_val)
print("Validation acc: ", acc)
```

```
Training acc:  0.427
Validation acc:  0.432
```

3 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%.

[15]:

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

```
Test accuracy:  0.364
```

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: Expanding the dataset for training purposes can assist in narrowing the gap between training and testing accuracy, as it provides the model with a more diverse range of examples to learn from, allowing for better generalization to new and unseen data. Similarly, augmenting the regularization strength can help decrease overfitting, thus resulting in a smaller disparity between the model's training and testing accuracy.