classification_nn

February 14, 2024

1 ECE 176 Assignment 4: 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.

[174]:
```
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from utils.data_processing import get_cifar10_data
from 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)
```

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

[175]:
```
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"]
```

[176]:
```
# Import more utilies and the layers you have implemented
from layers.sequential import Sequential
from layers.linear import Linear
from layers.relu import ReLU
from layers.softmax import Softmax
from layers.loss_func import CrossEntropyLoss
from utils.optimizer import SGD
from utils.dataset import DataLoader
from utils.trainer import Trainer
```

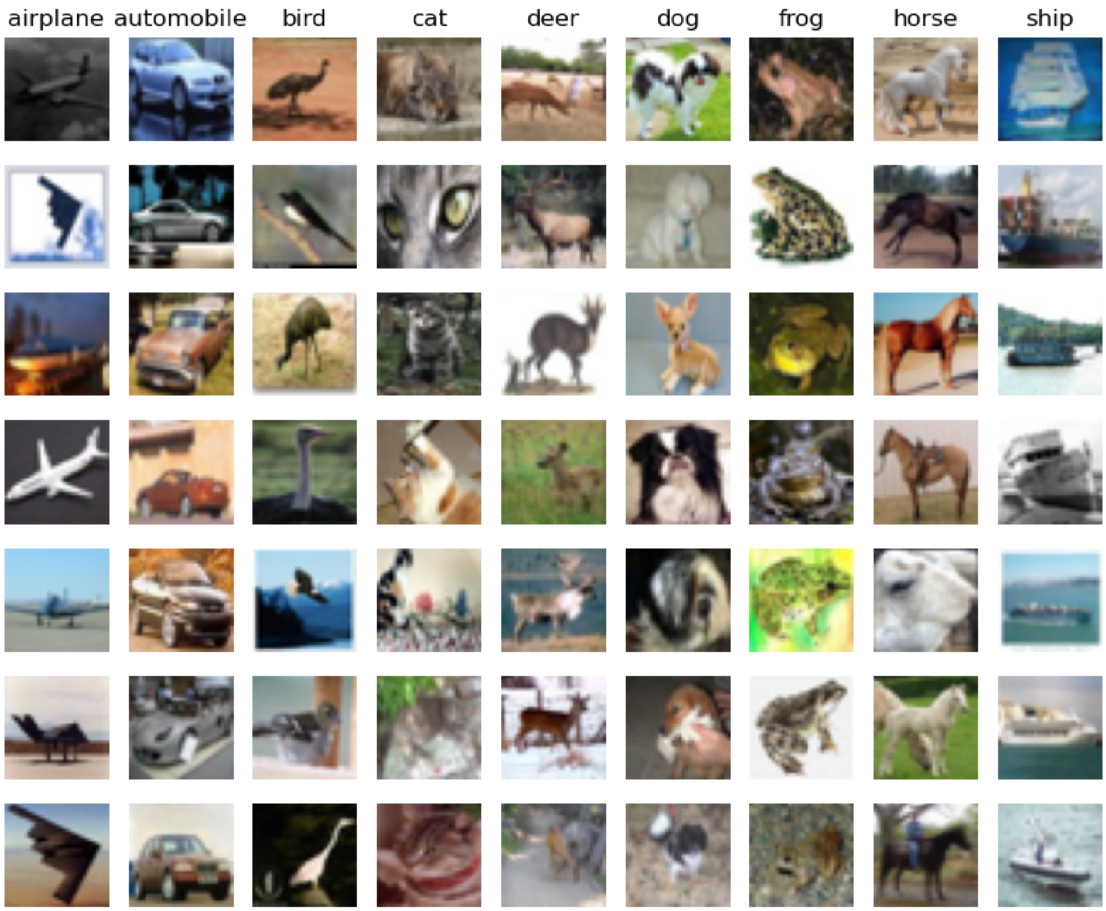
1.1 Visualize some examples from the dataset.

[177]:
```
# We show a few examples of training images from each class.
classes = [
    "airplane",
    "automobile",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
]
```

```
samples_per_class = 7


def visualize_data(dataset, classes, samples_per_class):
    num_classes = len(classes)
    for y, cls in enumerate(classes):
        idxs = np.flatnonzero(y_train == y)
        idxs = np.random.choice(idxs, samples_per_class, replace=False)
        for i, idx in enumerate(idxs):
            plt_idx = i * num_classes + y + 1
            plt.subplot(samples_per_class, num_classes, plt_idx)
            plt.imshow(dataset[idx])
            plt.axis("off")
            if i == 0:
                plt.title(cls)
    plt.show()


# Visualize the first 10 classes
visualize_data(
    x_train.reshape(5000, 3, 32, 32).transpose(0, 2, 3, 1),
    classes,
    samples_per_class,
)
```



1.2 Initialize the model

[196]:
```
input_size = 3072
hidden_size = 50  # 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])
```

[227]:
```
# 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=0.1, weight_decay=0.01)
loss_func = CrossEntropyLoss()
epoch = 150  # (Hyper-parameter)
batch_size = 200  # (Reduce the batch size if your computer is unable to handle␣
 ↪it)
```

[228]:
```
# Initialize the trainer class by passing the above modules
trainer = Trainer(
    dataset, optim, net, loss_func, epoch, batch_size, validate_interval=3
)
```

[229]:
```
# 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: 2.301742
Validate Acc: 0.084
Epoch Average Loss: 2.281892
Epoch Average Loss: 2.228768
Epoch Average Loss: 2.185306
Validate Acc: 0.128
Epoch Average Loss: 2.183483
Epoch Average Loss: 2.160374
Epoch Average Loss: 2.141034
Validate Acc: 0.180
Epoch Average Loss: 2.133053
Epoch Average Loss: 2.110589
Epoch Average Loss: 2.090175
Validate Acc: 0.240
Epoch Average Loss: 2.066796
Epoch Average Loss: 2.050382
Epoch Average Loss: 2.054510
Validate Acc: 0.252
Epoch Average Loss: 2.024010
Epoch Average Loss: 2.025530
Epoch Average Loss: 2.012800
Validate Acc: 0.304
Epoch Average Loss: 2.008170
Epoch Average Loss: 1.989260
Epoch Average Loss: 1.971304
```

```
Validate Acc: 0.288
Epoch Average Loss: 1.988027
Epoch Average Loss: 1.946396
Epoch Average Loss: 1.930983
Validate Acc: 0.288
Epoch Average Loss: 1.939809
Epoch Average Loss: 1.927701
Epoch Average Loss: 1.941387
Validate Acc: 0.284
Epoch Average Loss: 1.902646
Epoch Average Loss: 1.884240
Epoch Average Loss: 1.889275
Validate Acc: 0.320
Epoch Average Loss: 1.890581
Epoch Average Loss: 1.875126
Epoch Average Loss: 1.893421
Validate Acc: 0.324
Epoch Average Loss: 1.838587
Epoch Average Loss: 1.871209
Epoch Average Loss: 1.865429
Validate Acc: 0.324
Epoch Average Loss: 1.824595
Epoch Average Loss: 1.832390
Epoch Average Loss: 1.841185
Validate Acc: 0.352
Epoch Average Loss: 1.830221
Epoch Average Loss: 1.835322
Epoch Average Loss: 1.783136
Validate Acc: 0.348
Epoch Average Loss: 1.802317
Epoch Average Loss: 1.800245
Epoch Average Loss: 1.797675
Validate Acc: 0.364
Epoch Average Loss: 1.802647
Epoch Average Loss: 1.765599
Epoch Average Loss: 1.793254
Validate Acc: 0.380
Epoch Average Loss: 1.766292
Epoch Average Loss: 1.756402
Epoch Average Loss: 1.759399
Validate Acc: 0.344
Epoch Average Loss: 1.801209
Epoch Average Loss: 1.754717
Epoch Average Loss: 1.711059
Validate Acc: 0.324
Epoch Average Loss: 1.766152
Epoch Average Loss: 1.747408
Epoch Average Loss: 1.770996
```

```
Validate Acc: 0.312
Epoch Average Loss: 1.730554
Epoch Average Loss: 1.703038
Epoch Average Loss: 1.776465
Validate Acc: 0.316
Epoch Average Loss: 1.755103
Epoch Average Loss: 1.724518
Epoch Average Loss: 1.708838
Validate Acc: 0.320
Epoch Average Loss: 1.732780
Epoch Average Loss: 1.721135
Epoch Average Loss: 1.702676
Validate Acc: 0.352
Epoch Average Loss: 1.725721
Epoch Average Loss: 1.679664
Epoch Average Loss: 1.733167
Validate Acc: 0.356
Epoch Average Loss: 1.691008
Epoch Average Loss: 1.696754
Epoch Average Loss: 1.691202
Validate Acc: 0.388
Epoch Average Loss: 1.731397
Epoch Average Loss: 1.681860
Epoch Average Loss: 1.699882
Validate Acc: 0.316
Epoch Average Loss: 1.698635
Epoch Average Loss: 1.682454
Epoch Average Loss: 1.679310
Validate Acc: 0.340
Epoch Average Loss: 1.666018
Epoch Average Loss: 1.698617
Epoch Average Loss: 1.674649
Validate Acc: 0.392
Epoch Average Loss: 1.671622
Epoch Average Loss: 1.665590
Epoch Average Loss: 1.660494
Validate Acc: 0.344
Epoch Average Loss: 1.667547
Epoch Average Loss: 1.649367
Epoch Average Loss: 1.668671
Validate Acc: 0.360
Epoch Average Loss: 1.649116
Epoch Average Loss: 1.663863
Epoch Average Loss: 1.648925
Validate Acc: 0.320
Epoch Average Loss: 1.642471
Epoch Average Loss: 1.635059
Epoch Average Loss: 1.609347
```

```
Validate Acc: 0.396
Epoch Average Loss: 1.643278
Epoch Average Loss: 1.636681
Epoch Average Loss: 1.654829
Validate Acc: 0.380
Epoch Average Loss: 1.632910
Epoch Average Loss: 1.598240
Epoch Average Loss: 1.675084
Validate Acc: 0.368
Epoch Average Loss: 1.636089
Epoch Average Loss: 1.638425
Epoch Average Loss: 1.640596
Validate Acc: 0.440
Epoch Average Loss: 1.662195
Epoch Average Loss: 1.679255
Epoch Average Loss: 1.578373
Validate Acc: 0.372
Epoch Average Loss: 1.611059
Epoch Average Loss: 1.602660
Epoch Average Loss: 1.613588
Validate Acc: 0.348
Epoch Average Loss: 1.579914
Epoch Average Loss: 1.611923
Epoch Average Loss: 1.580626
Validate Acc: 0.384
Epoch Average Loss: 1.607162
Epoch Average Loss: 1.672608
Epoch Average Loss: 1.655440
Validate Acc: 0.396
Epoch Average Loss: 1.597725
Epoch Average Loss: 1.627872
Epoch Average Loss: 1.649725
Validate Acc: 0.404
Epoch Average Loss: 1.622628
Epoch Average Loss: 1.607590
Epoch Average Loss: 1.631695
Validate Acc: 0.416
Epoch Average Loss: 1.584724
Epoch Average Loss: 1.649931
Epoch Average Loss: 1.578023
Validate Acc: 0.388
Epoch Average Loss: 1.647135
Epoch Average Loss: 1.653026
Epoch Average Loss: 1.603506
Validate Acc: 0.392
Epoch Average Loss: 1.608388
Epoch Average Loss: 1.620798
Epoch Average Loss: 1.592218
```

```
Validate Acc: 0.368
Epoch Average Loss: 1.584916
Epoch Average Loss: 1.581621
Epoch Average Loss: 1.735042
Validate Acc: 0.352
Epoch Average Loss: 1.654244
Epoch Average Loss: 1.641456
Epoch Average Loss: 1.627247
Validate Acc: 0.428
Epoch Average Loss: 1.566804
Epoch Average Loss: 1.598903
Epoch Average Loss: 1.607328
Validate Acc: 0.392
Epoch Average Loss: 1.566543
Epoch Average Loss: 1.606603
Epoch Average Loss: 1.603045
Validate Acc: 0.368
Epoch Average Loss: 1.587894
Epoch Average Loss: 1.566260
Epoch Average Loss: 1.560839
Validate Acc: 0.400
Epoch Average Loss: 1.579264
Epoch Average Loss: 1.593444
Epoch Average Loss: 1.672665
Validate Acc: 0.400
Epoch Average Loss: 1.605165
Epoch Average Loss: 1.610822
Epoch Average Loss: 1.536899
Validate Acc: 0.360
Epoch Average Loss: 1.596438
Epoch Average Loss: 1.581373
```

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

[230]:
```python
from 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:  0.4788
Validation acc:  0.42
```

1.2.2 Debug the training

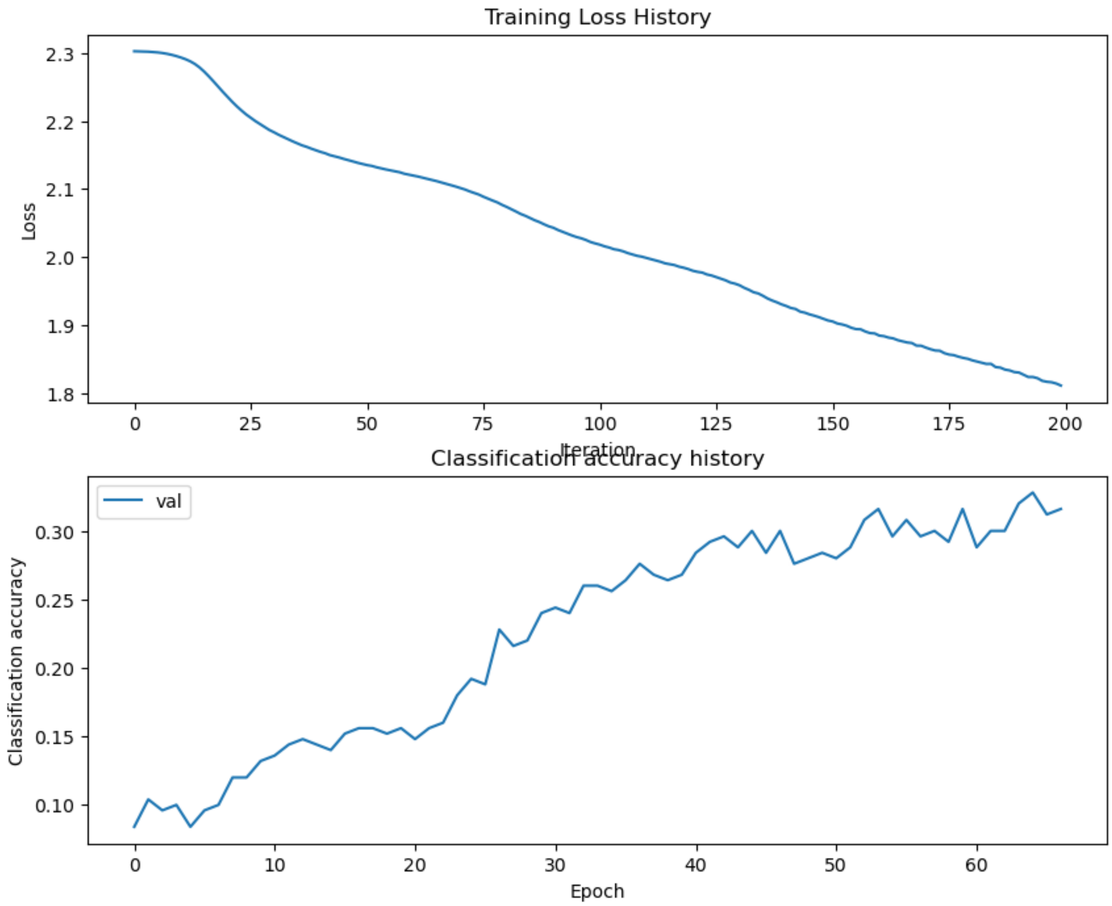
With the default parameters we provided above, you should get a validation accuracy of around 0.2~0.3 on the validation set. This isn't very good.

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.

[183]:
```
# 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()
```



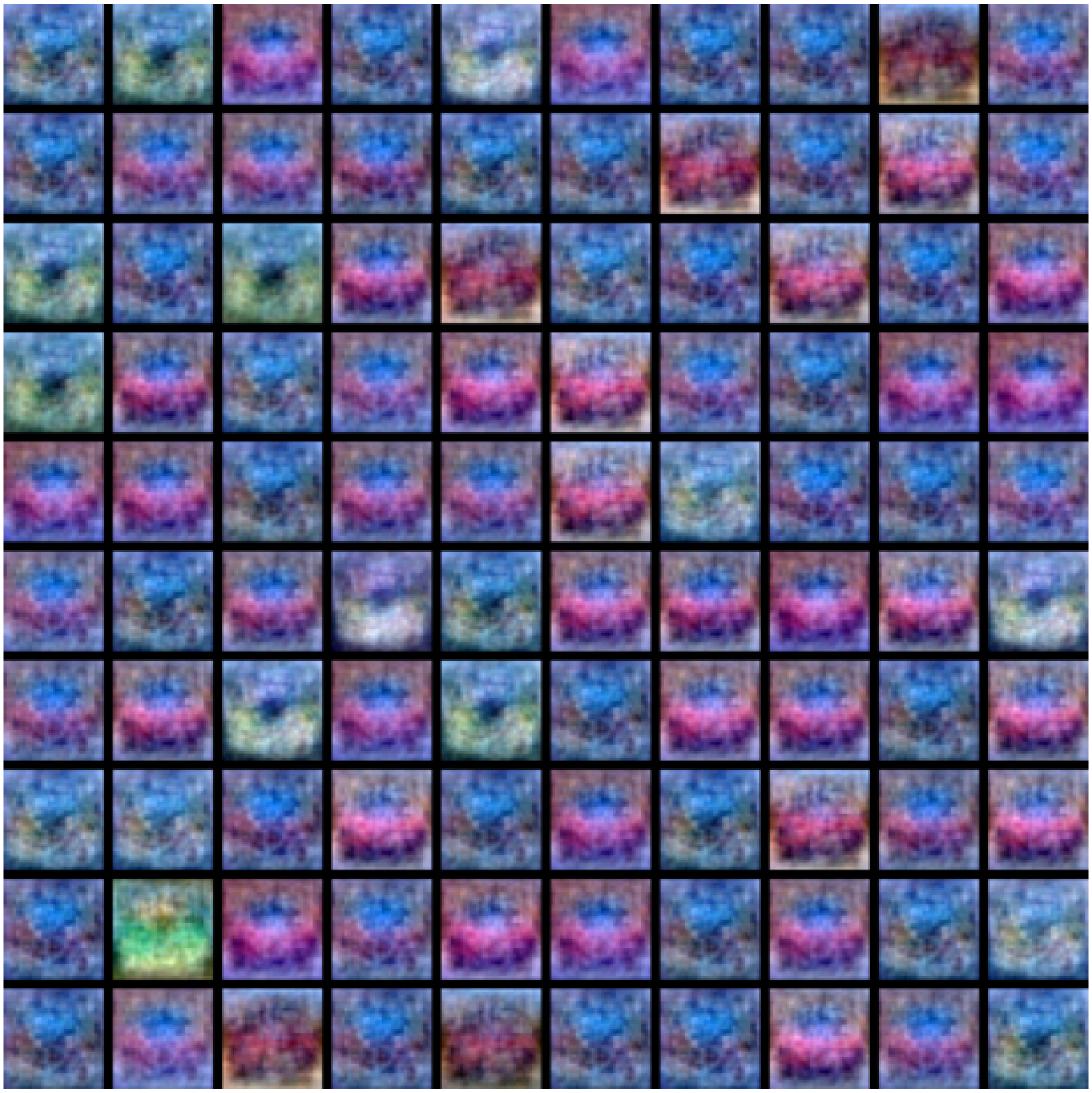
[184]:
```
from utils.vis_utils import visualize_grid

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

# Visualize the weights of the network

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


show_net_weights(net)
```



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: First I started to tune the hyperparameters of the model by decreasing the learning rate to 0.001 and I found out that the accuracy on the validation data set is lower than the default learning rate. Then I tried a higher learning rate 0.1 for the model and found that the accuracy on the validation training set increased. Next, I decreased the number of the epoch and the hidden_size to make the model train faster and surprisingly found out that the validation accuracy increased.

[237]:
```
best_net_hyperparams = [0.1, 0.01, 150, 50]  # store the best model into this
best_net = net

from utils.evaluation import get_classification_accuracy

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)

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

```
# TODO: Show the above plots and visualizations for the default params (already␣
↪#
# done) and the best hyper-params you obtain. You only need to show this for 2 ␣
↪#
# sets of hyper-params.                                                         ␣
↪#
# You just need to store values for the hyperparameters in best_net_hyperparams␣
↪#
# as a list in the order
# best_net_hyperparams = [lr, weight_decay, epoch, hidden_size]
################################################################################
```

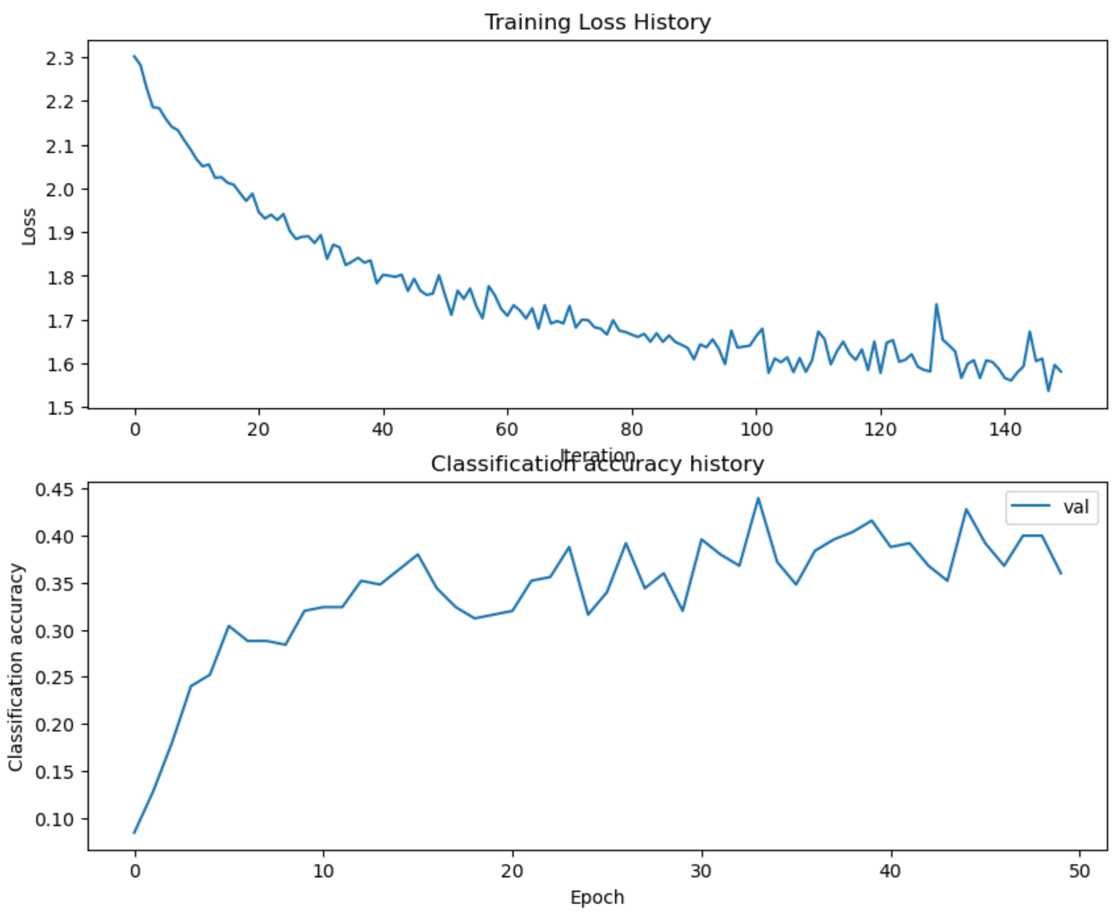
```
Training acc:  0.4788
Validation acc:  0.42
```

[236]:

```
# TODO: Plot the training_error and validation_accuracy of the best network (5%)
# 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(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%)
best_net_hyperparams = [0.1, 0.01, 150, 50]
```



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%.

[233]:
```
test_acc = (best_net.predict(x_test) == y_test).mean()
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.352
```

Inline Question (9%) 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: Training on a larger dataset and Increase the regularization strenth

Your Explanation: Train the model on a larger dataset can improve the generalization of the model and prevent overfitting on the dataset which help us to reduce the gap between the training set and testing set. Increasing the regularization strenth can also help the model to prevent overfitting which again reduce the gap between the training set and testing set.

3.1 Survey (1%)

3.1.1 Question:

How many hours did you spend on this assignment?

3.1.2 Your Answer: 2 hours