

Neural Networks and Deep Learning HW #3

1. Given $x \in \mathbb{R}^n$, $W \in \mathbb{R}^{m \times n}$ where $m < n$.

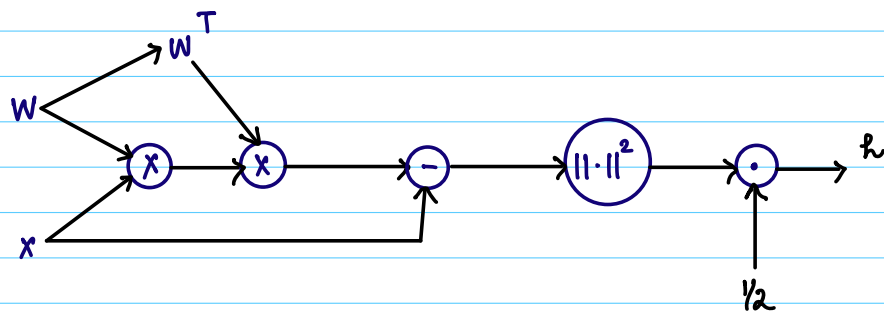
a) First let's consider the loss equation

$$L = \frac{1}{2} \|W^T W x - x\|^2$$

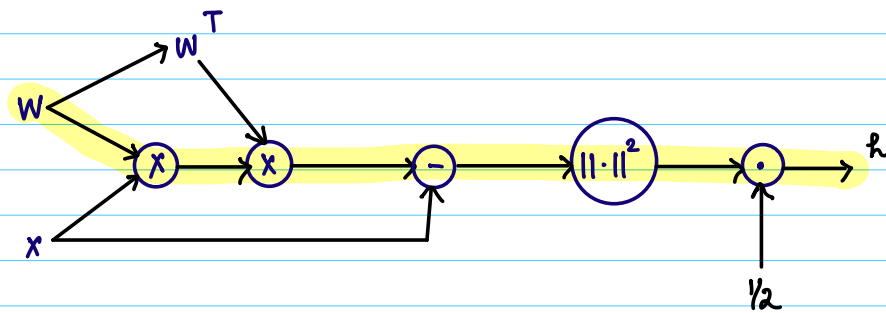
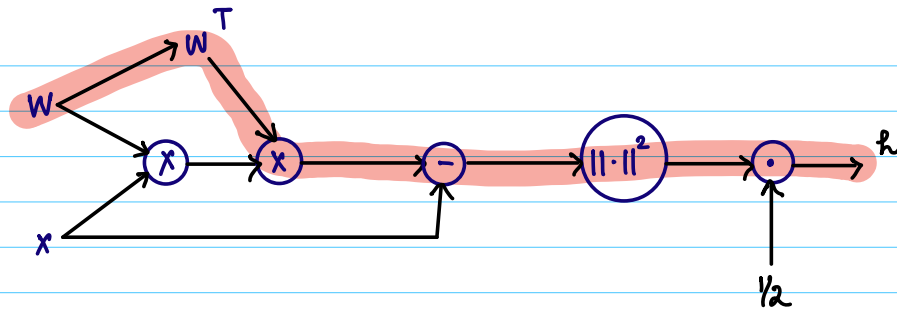
* We know that the loss L should be minimized, i.e. $W^T W x - x$ must be minimized.

* Wx is the hidden representation and for the loss to be minimized, Wx will have to preserve information about x .

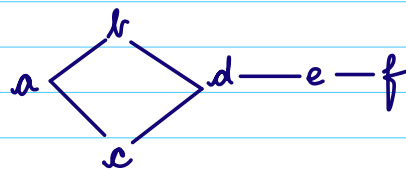
b) Computation graph:-



c) The two paths to W are highlighted below:



Since W contributes to two paths, the ∇_W^h will be expressed as a sum of derivatives along each path. Let's consider this case:



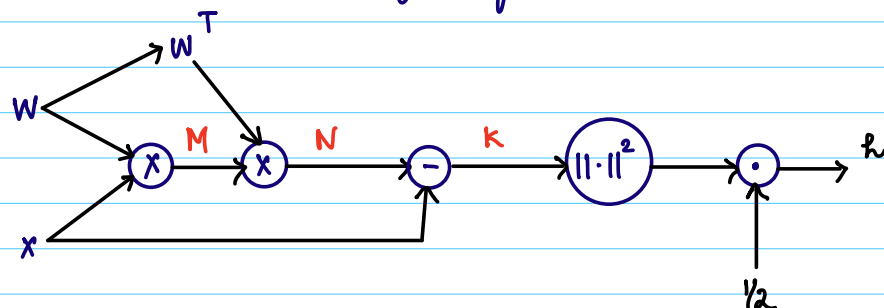
This graph represents the above scenario and consists of two paths to reach a .

$$a \rightarrow b \rightarrow d \rightarrow e \rightarrow f$$

$$a \rightarrow c \rightarrow d \rightarrow e \rightarrow f$$

$$\begin{aligned} \text{hence } \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} \cdot \frac{\partial b}{\partial a} \\ &\quad + \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial c} \cdot \frac{\partial c}{\partial a} \end{aligned}$$

d) Let's consider the computation graph from 8):



* $K = W^T W x - x$
 where $W^T = n \times m$
 $W = m \times n$
 $x = n \times 1$

Hence $K = W^T W x - x \in \mathbb{R}^n$

* $N = W^T M$
 $N = W^T W x$
 where
 $W^T = n \times m$
 $W x = m \times 1$
 Hence $N = W^T M \in \mathbb{R}^n$

* $M = W x$
 where $W = m \times n$
 $x = n \times 1$
 Hence $M = W x \in \mathbb{R}^m$

* $h = \frac{1}{2} \|K\|^2$, $\partial h / \partial K = K \rightarrow \textcircled{0}$

* $K = N - x$, $\partial h / \partial N = \partial K / \partial N \times \partial h / \partial K$ (chain rule)

We know that $\partial h / \partial K = K$

$\partial K / \partial N = \partial (N - x) / \partial N = I$

Hence $\partial h / \partial N = K \rightarrow \textcircled{1}$

We could also say that $\partial h / \partial K = \partial h / \partial N$ as $-$ operator distributes the gradient.

$$* \quad N = W^T M \quad \text{where } M = Wx \in \mathbb{R}^m$$

$$\partial h / \partial W^T = \partial h / \partial N \cdot \partial N / \partial W^T$$

$$\partial h / \partial N = k \in \mathbb{R}^n$$

$$\partial N / \partial W^T = \partial (W^T M) / \partial W^T = M^T = (Wx)^T \in \mathbb{R}^{1 \times m}$$

$$\text{hence } \partial h / \partial W^T = k (Wx)^T \longrightarrow \textcircled{2}$$

$$* \quad \partial h / \partial M = \partial h / \partial N \cdot \partial N / \partial M$$

$$\partial h / \partial N = k \in \mathbb{R}^n$$

$$\text{we know that } M = Wx \in \mathbb{R}^m, \text{ hence } \partial h / \partial M \in \mathbb{R}^m$$

$$\partial N / \partial M = \partial (W^T M) / \partial M = W \in \mathbb{R}^{m \times n}$$

$$\text{hence } \partial h / \partial M = Wk$$

$$* \quad \partial h / \partial W = \partial h / \partial M \times \partial M / \partial W$$

$$\partial h / \partial W \text{ must be } \mathbb{R}^{m \times n}$$

$$\partial h / \partial M = Wk \in \mathbb{R}^m, \quad \partial M / \partial W = \partial (Wx) / \partial W = x^T \in \mathbb{R}^{1 \times n}$$

$$\therefore \partial h / \partial W = WKX^T \in \mathbb{R}^{m \times n} \longrightarrow (3)$$

From (2) and (3), we know both contribute to ∇_W^h i.e

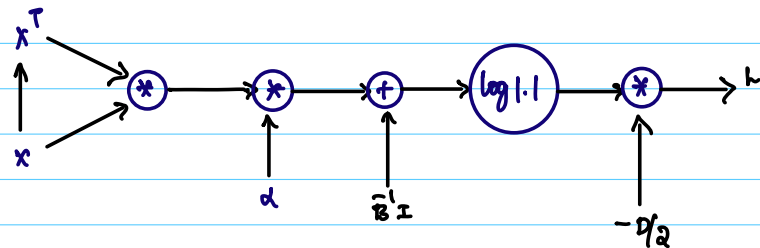
$$\nabla_W^h = \text{Backprop of } W + \text{Backprop of } W^T$$

Hence

$$\nabla_W^h = WKX^T + K(WX)^T = WKX^T + WXK^T$$

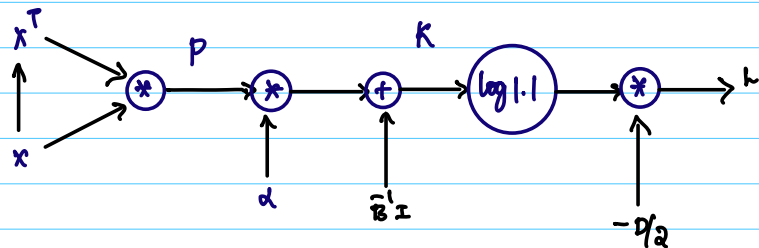
Question 2:

a) Computation graph for h_1 :



b) We need to compute $\partial h_1 / \partial x$

From the question, we know that $K = \alpha x x^T + \beta^{-1} I$



$$h_1 = -D/2 \log(|K|)$$

$$\partial h_1 / \partial K = -D/2 (K^{-1})^T = -D/2 (K^T)^{-1} \quad (\text{From the Matrix cookbook})$$

Now, we propagate backwards, encountering a \oplus operation,

we know \oplus distributes the gradients and hence the gradient remains the same.

Now, we encounter a \otimes operator, hence the gradient @ P would be:

$$\partial h / \partial P = -\alpha D/2 \cdot (K^T)^{-1}$$

$$\partial h / \partial x x^T = -\alpha D/2 \cdot (K^T)^{-1}$$

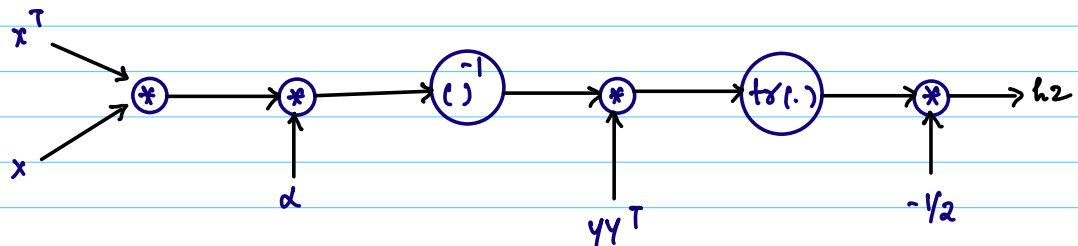
$$P = x x^T$$

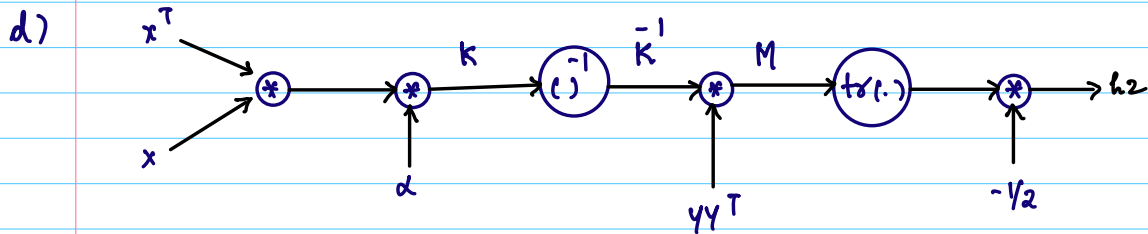
$$\partial h / \partial x = \partial P / \partial x \cdot \partial h / \partial P$$

$$\partial P / \partial x = x x^T = 2x$$

$$\therefore \partial h / \partial x = -\alpha D (K^T)^{-1} x$$

c) Computational graph for h_2 :





$$h_2 = -1/2 \operatorname{tr} ((\alpha x x^T + \bar{K}^{-1} \mathbf{I})^{-1} y y^T)$$

$$h_2 = -1/2 \operatorname{tr} (M)$$

$$\partial h_2 / \partial M = -1/2 \mathbf{I} \quad (\text{From Matrix Cookbook})$$

$$\partial h_2 / \partial \bar{K}^{-1} = -1/2 \cdot y y^T$$

$$\partial h_2 / \partial K = 1/2 \bar{K}^{-1} y y^T \bar{K}^{-1}$$

Hence from the last part, we could say that:

$$\partial h_2 / \partial x = \alpha \bar{K}^{-1} y y^T \bar{K}^{-1} x$$

e) Hence $\partial h / \partial x = \partial h_1 / \partial x + \partial h_2 / \partial x$

$$\partial h / \partial x = \alpha \bar{K}^{-1} y y^T \bar{K}^{-1} x - \alpha D K^{-1} x$$

This is the 2-layer neural network notebook for ECE C147/C247 Homework #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the notebook entirely when completed.

The goal of this notebook is to give you experience with training a two layer neural network.

```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

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

Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass

```
from mndl.neural_net import TwoLayerNet
```

```
# 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
num_inputs = 5

def init_toy_model():
    np.random.seed(1)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.zeros(1)
    return X, y

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

Compute forward pass scores

```
# Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores')
correct_scores = np.asarray([
    [-1.07260209, 0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908, 0.15259725, -0.39578548],
    [-0.38172726, 0.10835902, -0.17328274],
    [-0.64417314, 0.18886813, -0.41106892]])
print(correct_scores)
print()

# 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:

```
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314  0.18886813 -0.41106892]]
```

correct scores:

```
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314  0.18886813 -0.41106892]]
```

Difference between your scores and correct scores:

```
3.3812119573975e-08
```

Forward pass loss

```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817
```

```
# should be very small, we get < 1e-12
print('Loss:', loss)
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

Loss: 1.071696123862817
Difference between your loss and correct loss: 0.0

Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

```
from utils.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be very small
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)
    param_grad_num = eval_numerical_gradient(f, net.params(param_name), verbose=False,
        print_=(''))
    max_relative_error = (1/len(param_grad_num)) * sum([abs(x) for x in param_grad_num])

W2 max relative error: 2.063223346013647e-10
b2 max relative error: 1.2482633693659668e-09
W1 max relative error: 1.28328951808708e-09
b1 max relative error: 3.172680285697327e-09
```

Training the network

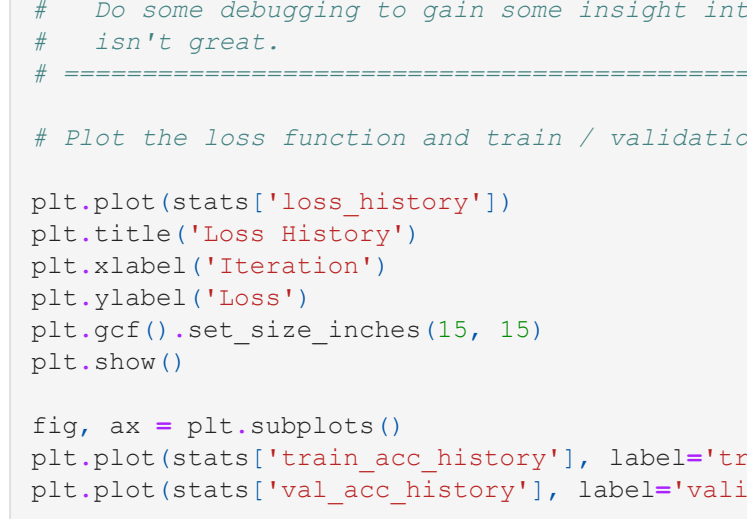
Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax.

```
net = init_toy_model()
stats = net.train(X, y, X_val, y_val,
    learning_rate=1, reg=0.05,
    num_iters=10, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('training loss history')
plt.show()
```

Final training loss: 0.014497864587765906



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

```
from utils.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    # it for the two-layer neural net classifier.
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cifar10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    y_val = y_train[mask]
    mask = list(range(num_validation, num_validation + num_test))
    y_test = y_train[mask]
    mask = list(range(num_test, num_test + num_test))
    y_test = y_train[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    y_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)
    return X_train, X_train, X_val, y_val, X_test, y_test

# Invoke the above function to get our data.
X_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

Train data shape: (49000, 3072)
Train labels shape: (49000,)
Validation data shape: (1000, 3072)
Validation labels shape: (1000,)
Test data shape: (1000, 3072)
Test labels shape: (1000,)

Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
    num_iters=100, batch_size=20,
    learning_rate=1e-4, learning_rate_decay=0.95,
    reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)

# Save this net as the variable subopt_net for later comparison.
subopt_net = net

iteration 0 / 1000: loss 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.2518259043164135
iteration 400 / 1000: loss 2.18899523546776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.9801688621081942
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy: 0.283
```

Questions:

The training accuracy isn't great.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

```
stats['train_acc_history']
```

```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

```
stats['val_acc_history']
```

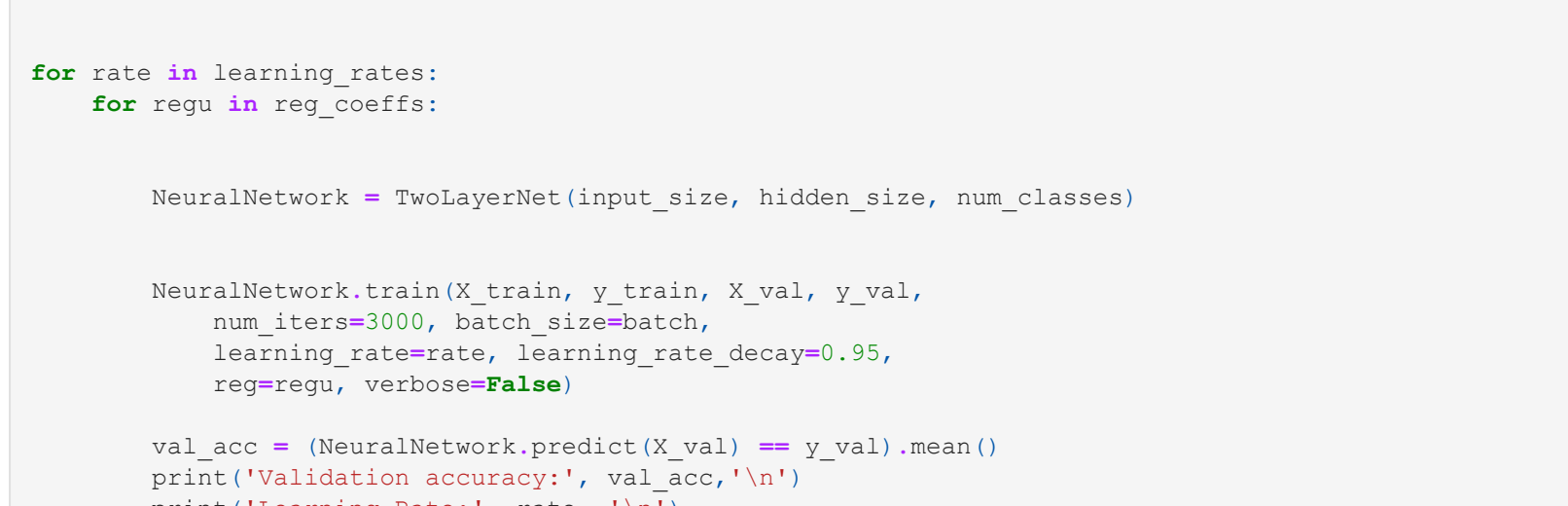
```
[0.115, 0.177, 0.208, 0.26, 0.285]
```

```
# YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.gca().set_size_inches(15, 15)
plt.show()

fig, ax = plt.subplots()
plt.plot(stats['train_acc_history'], label='training')
plt.plot(stats['val_acc_history'], label='validation')
legend = ax.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.80')

plt.title('History of classification accuracy')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend(loc='lower right')
plt.gca().set_size_inches(15, 12)
plt.show()

# END YOUR CODE HERE
```



Answers:

(1) From the first plot, we can say that the Loss for the first 200 iterations is the same and does not decrease. Post that, we can observe that the Loss decreases linearly which suggests that the Learning Rate used is not large enough. Secondly, from the Accuracy plot, we observe that for the first 1000 iterations, both the training and validation accuracy seems to increase and do not converge. This might suggest that we need to increase the number of iterations to make the model converge.

(2) The Learning Rate must be increased to ensure the Loss drops exponentially rather than linearly. Also, the number of iterations must be increased to ensure model convergence.

Optimize the neural network

Use the following np part of the `TwoLayerNet` to optimize your hyperparameters on the validation set. Store your nets as `best_net`.

```
best_net = None # store the best model into this

# YOUR CODE HERE
# Optimize over your hyperparameters to arrive at the best neural
# network. You should be able to get over 50% validation accuracy.
# For this part of the notebook, we will give credit based on the
# accuracy you get. Your score on this question will be multiplied by:
# min(floor((k - 28)/3) / 42, 1)
# where if you get 50% or higher validation accuracy, you get full
# points.
# Note, you need to use the same network structure (keep hidden size = 50)!
# =====
best_acc = -1
learning_rates = [1e-1, 5e-2, 3e-2, 1e-2, 5e-3, 3e-3, 1e-3]
results = []
for lr in learning_rates:
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
        num_iters=1000, batch_size=20,
        learning_rate=lr, learning_rate_decay=0.95,
        reg=0.35)
    y_train_pred = net.predict(X_train)
    acc_train = np.mean(y_train == y_train_pred)
    y_val_pred = net.predict(X_val)
    acc_val = np.mean(y_val == y_val_pred)
    results.append((acc_train, acc_val))

    if best_acc < acc_val:
        best_acc = acc_val
        best_net = net

# Print out results
for lr in sorted(learning_rates):
    train_acc, val_acc, results = results[results.index:]
    print('Learning rates: %f Train accuracy: %f Validation accuracy: %f' % (lr, train_acc, val_acc))

print('Best validation accuracy: %f' % best_acc)
```

```
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.title('Classification Accuracy History')
plt.xlabel('iteration')
plt.ylabel('Accuracy')
plt.show()

# =====
# END YOUR CODE HERE
# Do some debugging to gain some insight into why the optimization
# isn't great.
# Plot the loss function and train / validation accuracies
plt.plot(stats['loss_history'])
plt
```



```

import numpy as np
import matplotlib.pyplot as plt

class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network. The net has an input dimension of
    N, a hidden layer dimension of H, and performs classification over C classes.
    We train the network with a softmax loss function and L2 regularization on the
    weight matrices. The network uses a ReLU nonlinearly after the first fully
    connected layer.

    In other words, the network has the following architecture:

    input -> fully connected layer -> ReLU -> fully connected layer -> softmax

    The outputs of the second fully-connected layer are the scores for each class.
    """

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        """
        Initialize the model. Weights are initialized to small random values and
        biases are initialized to zero. Weights and biases are stored in the
        variable self.params, which is a dictionary with the following keys:

        W1: First layer weights; has shape (H, D)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (C, H)
        b2: Second layer biases; has shape (C,)

        Inputs:
        - input_size: The dimension D of the input data.
        - hidden_size: The number of neurons H in the hidden layer.
        - output_size: The number of classes C.
        """
        self.params = {}
        self.params['W1'] = std * np.random.randn(hidden_size, input_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = std * np.random.randn(output_size, hidden_size)
        self.params['b2'] = np.zeros(output_size)

    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural
        network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
          an integer in the range 0 <= y[i] < C. This parameter is optional; if it
          is not passed then we only return scores, and if it is passed then we
          instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, instead return a tuple of:
        - loss: loss (data loss and regularization loss) for this batch of training
          samples.
        - grads: Dictionary mapping parameter names to gradients of those parameters
          with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

        # Compute the forward pass
        scores = None

        # =====
        # YOUR CODE HERE:
        # Calculate the output scores of the neural network. The result
        # should be (N, C). As stated in the description for this class,
        # there should not be a ReLU layer after the second FC layer.
        # The output of the second FC layer is the output scores. Do not
        # use a for loop in your implementation.
        # =====

        relu = lambda x: np.maximum(0, x)
        H1 = relu(W1.dot(X) + b1) # X is (5,4) W1.T (4,10) -> (5,10)
        scores = H1.dot(W2) + b2 # H1 is (5,10), W2 is (10,10) -> (5,10)

        # =====
        # END YOUR CODE HERE
        # =====

        # If the targets are not given then jump out, we're done
        if y is None:
            return scores

        # Compute the loss
        loss = None

        # =====
        # YOUR CODE HERE:
        # Calculate the loss of the neural network. This includes the
        # softmax loss and the L2 regularization for W1 and W2. Store the
        # total loss in the variable loss. Multiply the regularization
        # loss by 0.5 (in addition to the factor reg).
        # =====

        # scores is num_examples by num_classes
        class_prob = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)
        prob_correct_y = class_prob[np.arange(N), y]
        log_loss = -np.log(prob_correct_y)
        sum_log_loss = np.sum(log_loss)
        loss = sum_log_loss / N

        frob_norm_w1 = np.sum(W1**2)
        frob_norm_w2 = np.sum(W2**2)
        reg_w1 = 0.5 * reg * frob_norm_w1
        reg_w2 = 0.5 * reg * frob_norm_w2
        regularized_loss = reg_w1 + reg_w2

        loss += regularized_loss

        #reg_loss = 0.5 * reg * (np.linalg.norm(W1, 'fro')**2 + np.linalg.norm(W2, 'fro')**2)
        #loss = (np.sum(-np.log(np.exp(scores[np.arange(N), y]) / np.sum(np.exp(scores), axis = 1)))) / N + reg_loss

        # =====
        # END YOUR CODE HERE
        # =====

        grads = {}

        # =====
        # YOUR CODE HERE:
        # Implement the backward pass. Compute the derivatives of the
        # weights and the biases. Store the results in the grads
        # dictionary. e.g., grads['W1'] should store the gradient for
        # W1, and be of the same size as W1.
        # =====

        update_scores = class_probabilities
        update_scores[np.arange(N), y] -= 1
        update_scores /= N

        grads['W2'] = np.dot(H1.T, update_scores).T
        grads['b2'] = np.sum(update_scores, axis=0)
        dH2 = np.dot(update_scores, W2)

        dLdA = dH2
        dLdA[H1 <= 0] = 0

        grads['W1'] = np.dot(dLdA.T, X)
        grads['b1'] = np.sum(dLdA, axis=0)

        grads['W2'] += reg * W2
        grads['W1'] += reg * W1

        # =====
        # END YOUR CODE HERE
        # =====

        return loss, grads

    def train(self, X, y, X_val, y_val,
              learning_rate=1e-3, learning_rate_decay=0.95,
              reg=5e-5, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this neural network using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) giving training data.
        - y: A numpy array of shape (N,) giving training labels; y[i] = c means that
          X[i] has label c, where 0 <= c < C.
        - X_val: A numpy array of shape (N_val, D) giving validation data.
        - y_val: A numpy array of shape (N_val,) giving validation labels.
        - learning_rate: Scalar giving learning rate for optimization.
        - learning_rate_decay: Scalar giving factor used to decay the learning rate
          after each epoch.
        - reg: Scalar giving regularization strength.
        - num_iters: Number of steps to take when optimizing.
        - batch_size: Number of training examples to use per step.
        - verbose: boolean; if true print progress during optimization.
        """
        num_train = X.shape[0]
        iterations_per_epoch = max(num_train / batch_size, 1)

        # Use SGD to optimize the parameters in self.model
        loss_history = []
        train_acc_history = []
        val_acc_history = []

        for it in np.arange(num_iters):
            X_batch = None
            y_batch = None

            # =====
            # YOUR CODE HERE:
            # Create a minibatch by sampling batch_size samples randomly.
            # =====

            random_indices = np.random.choice(np.arange(num_train), batch_size)
            X_batch = X[random_indices]
            y_batch = y[random_indices]

            # =====
            # END YOUR CODE HERE
            # =====

            # Compute loss and gradients using the current minibatch
            loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
            loss_history.append(loss)

            # =====
            # YOUR CODE HERE:
            # Perform a gradient descent step using the minibatch to update
            # all parameters (i.e., W1, W2, b1, and b2).
            # =====

            self.params['W2'] -= learning_rate * grads['W2']
            self.params['b2'] -= learning_rate * grads['b2']
            self.params['W1'] -= learning_rate * grads['W1']
            self.params['b1'] -= learning_rate * grads['b1']

            # =====
            # END YOUR CODE HERE
            # =====

            if verbose and it % 100 == 0:
                print('iteration %d / %d: loss %f' % (it, num_iters, loss))

            # Every epoch, check train and val accuracy and decay learning rate.
            if it % iterations_per_epoch == 0:
                # Check accuracy
                train_acc = (self.predict(X_batch) == y_batch).mean()
                val_acc = (self.predict(X_val) == y_val).mean()
                train_acc_history.append(train_acc)
                val_acc_history.append(val_acc)

                # Decay learning rate
                learning_rate *= learning_rate_decay

        return {
            'loss_history': loss_history,
            'train_acc_history': train_acc_history,
            'val_acc_history': val_acc_history,
        }

    def predict(self, X):
        """
        Use the trained weights of this two-layer network to predict labels for
        data points. For each data point we predict scores for each of the C
        classes, and assign each data point to the class with the highest score.

        Inputs:
        - X: A numpy array of shape (N, D) giving N D-dimensional data points to
          classify.

        Returns:
        - y_pred: A numpy array of shape (N,) giving predicted labels for each
          of the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
          to have class c, where 0 <= c < C.
        """
        y_pred = None

        # =====
        # YOUR CODE HERE:
        # Predict the class given the input data.
        # =====

        num_examples = X.shape[0]
        y_pred = np.empty((num_examples, ), dtype=int)
        H1_preact = np.dot(X, self.params['W1'].T) + self.params['b1']

        #Apply ReLU
        H1 = np.maximum(0, H1_preact)

        #second layer
        H2 = np.dot(H1, self.params['W2'].T) + self.params['b2']

        #Apply softmax
        softmax = np.exp(H2) / np.sum(np.exp(H2), axis=1, keepdims=True)

        for i in range(num_examples):
            max_index = np.argmax(softmax[i])
            y_pred[i] = max_index

        # =====
        # END YOUR CODE HERE
        # =====

        return y_pred

```


Fully connected networks

In the previous notebook you implemented a simple two-layer neural network class. However, this class is not modular. If you wanted to change the number of layers, you would need to write a new loss and gradient function. If you wanted to optimize the network with different optimizers, you'd need to write new training functions. If you wanted to incorporate regularizations, you'd have to modify the loss and gradient function.

Instead of having to modify functions each time, for the rest of the class, we'll work in a more modular framework where we define forward and backward layers that calculate losses and gradients respectively. Since the forward and backward layers share intermediate values that are useful for calculating both the loss and the gradient, we'll also have these function return "caches" which store useful intermediate values.

The goal is that through this modular design, we can build different sized neural networks for various applications.

In this HW #3, we'll define the basic architecture, and in HW #4, we'll build on this framework to implement different optimizers and regularizations (like BatchNorm and Dropout).

Modular layers

This notebook will build out the forward and backward passes in the following manner. First, there will be a forward pass for a given layer with inputs (\mathbf{x}) and return the output of that layer (\mathbf{out}) as well as cached variables (\mathbf{cache}) that will be used to calculate the gradient in the backward pass.

```
def layer_forward(x, w):
    """ Receive inputs x and weights w """
    # Some intermediate value
    z = # ... some intermediate value
    # Do some more computations ...
    out = # the output

    cache = (x, w, z, out) # Values we need to compute gradients

    return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
    """
    Receive derivative of loss with respect to outputs and cache,
    and compute derivatives with respect to inputs.
    """
    # Unpack cache values
    x, w, z, out = cache

    # Use values in cache to compute derivatives
    dx = # Derivative of loss with respect to x
    dw = # Derivative of loss with respect to w

    return dx, dw
```

```
In [1]: ## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nnutils.fc_net import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from utils.solver import Solver

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

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
In [2]: # Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
    print('({})'.format(k, data[k].shape))

X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

Linear layers

In this section, we'll implement the forward and backward pass for the linear layers.

The linear layer forward pass is the function `affine_forward` in `nnutils/layers.py` and the backward pass is `affine_backward`.

After you have implemented these, test your implementation by running the cell below.

Affine layer forward pass

Implement `affine_forward` and then test your code by running the following cell.

```
In [3]: # Test the affine_forward function

num_inputs = 2
input_shape = (4, 3, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

w = np.linspace(-0.1, 0.1, numinput_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, numweight_size).reshape(np.prod(input_shape), output_dim)
b = np.linspace(-0.3, 0.1, numoutput_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([1.49834967, 1.70660132, 1.91485297],
                        [ 3.25553199, 3.914327, 3.7727342])

# Compare your output with ours. The error should be around 1e-9.
print('Testing affine_forward function:')
print('Difference: {}'.format(rel_error(out, correct_out)))

Testing affine_forward function:
Difference: 3.76800049986e-10
```

Affine layer backward pass

Implement `affine_backward` and then test your code by running the following cell.

```
In [4]: # Test the affine_backward function

x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], b, dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around 1e-10
print('Testing affine_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))

Testing affine backward function:
dx error: 8.393418893437644e-11
dw error: 5.78933207031032e-11
db error: 4.39392697894e-11
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

Implement the `relu_forward` function in `nnutils/layers.py` and then test your code by running the following cell.

```
In [5]: # Test the relu_forward function

input_shape = (-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([0., 0., 0., 0., ],
                        [ 0., 0., 0.0454545, 0.13636364, ],
                        [ 0.22727273, 0.31618182, 0.40909091, 0.5, ])

# Compare your output with ours. The error should be around 1e-8
print('Testing relu_forward function:')
print('Difference: {}'.format(rel_error(out, correct_out)))

Testing relu_forward function:
Difference: 4.9999798021158e-08
```

ReLU backward pass

Implement the `relu_backward` function in `nnutils/layers.py` and then test your code by running the following cell.

```
In [6]: x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)
_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 1e-12
print('Testing relu_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))

Testing relu backward function:
dx error: 3.27562444787486e-12
```

Combining the affine and ReLU layers

Often times an affine layer will be followed by a ReLU layer. So let's make one that puts them together. Layers that are combined are stored in `nnutils/layer_utils.py`.

Affine-ReLU layers

We've implemented `affine_relu_forward()` and `affine_relu_backward` in `nnutils/layer_utils.py`. Take a look at them to make sure you understand what's going on. Then run the following cell to ensure its implemented correctly.

```
In [7]: from nnutils.layer import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], b, dout)

print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))

Testing affine_relu forward and affine_relu backward:
dx error: 2.21764296235752e-10
dw error: 2.613940810927e-09
db error: 7.826729316297815e-12
```

Softmax losses

You've already implemented it, so we have written it in `layers.py`. The following code will ensure its working correctly.

```
In [8]: num_classes, num_inputs = 10, 50
w = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('Testing softmax_loss:')
print('Loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))

Testing softmax loss:
loss: 2.302599297115487
dx error: 6.65901943244728e-09
```

Implementation of a two-layer NN

In `nnutils/net.py`, implement the class `TwoLayerNet` which uses the layers you made here. When you have finished, the following cell will test your implementation.

```
In [9]: N, D, H, C = 3, 5, 50, 7
x = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-2
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ...')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ...')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
x_num = np.linspace(-0.5, 0.5, num=N*D).reshape(D, N, T)
scores = model.loss(x)
[correct_scores = np.asarray([
11.53165109, 12.2917344, 13.05819771, 13.81361102, 14.57386424, 15.33204765, 16.09215061,
12.05760081, 12.76934105, 13.47459113, 14.12304412, 14.81149128, 15.49994135, 16.18839143,
12.58373087, 13.2054771, 13.81736455, 14.43418138, 15.05099822, 15.66781506, 16.2846319 ])]
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-5, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
# It will with the Solver. Choose hyperparameters so that your validation
# accuracy is at least 50%. We won't have you optimize this further
# since you did it in the previous notebook.

model.loss(x) - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(x, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
    print('Running numeric gradient check with reg = {}'.format(reg))
    model.reg = reg
    loss, grads = model.loss(x, y)
    for name in sorted(grads):
        f = lambda: model.loss(x, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
        print('({}) relative error: {}'.format(name, rel_error(grad_num, grads[name])))

Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.289701627275205e-06
W2 relative error: 2.812817056147554e-06
b1 relative error: 1.42404896128542e-06
b2 relative error: 1.72889405818046e-09
W2 relative error: 4.329128523396134e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 1.52791528613156e-07
W2 relative error: 1.367835772201513e-07
b1 relative error: 1.536680149611563e-08
b2 relative error: 1.038961155678095e-10
```

Solver

We will now use the `utils Solver` class to train these networks. Familiarize yourself with the API in `utils/solver.py`. After you have done so, declare an instance of a `TwoLayerNet` with 200 units and then train it with the Solver. Choose parameters so that your validation accuracy is at least 50%.

```
In [10]: model = TwoLayerNet()
solver = None

# YOUR CODE HERE:
# - Declare an instance of a TwoLayerNet and then train
# - It will with the Solver. Choose hyperparameters so that your validation
# - accuracy is at least 50%. We won't have you optimize this further
# - since you did it in the previous notebook.

# =====

model = TwoLayerNet(hidden_dim=200, reg = 0.3)
solver = Solver(model, data,
                update_rule='sgd',
                learning_rate=8.5e-4,
                lr_decay=0.95,
                num_epochs=10, batch_size=25,
                print_every=100)
solver.train()

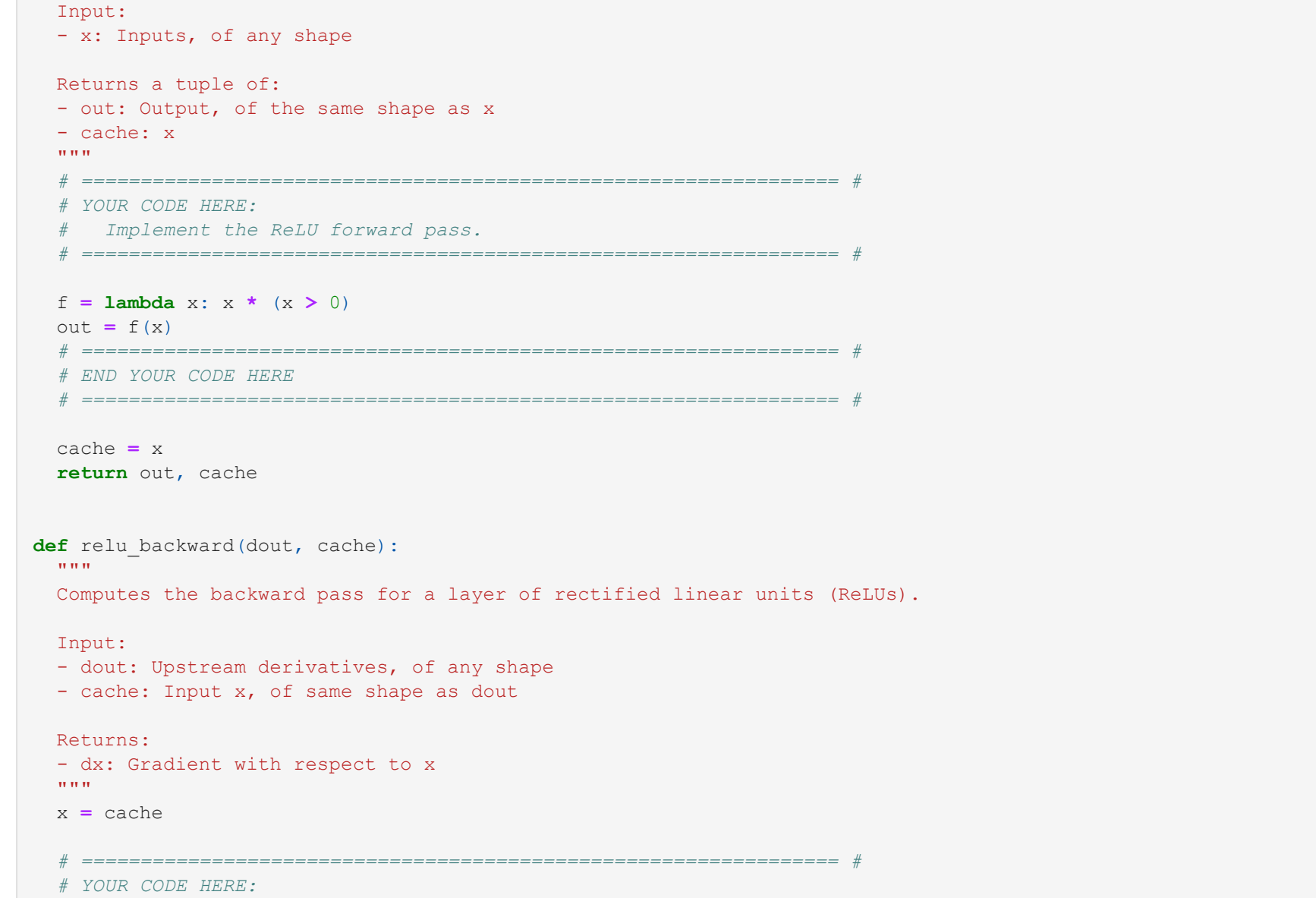
# =====
# END YOUR CODE HERE

(iteration 1 / 2770) loss: 2.366164
(Epoch 0 / 10) train acc: 0.127000; val_acc: 0.123000
(iteration 101 / 2770) loss: 1.48602
(iteration 201 / 2770) loss: 1.817794
(Epoch 1 / 10) train acc: 0.440000; val_acc: 0.429000
(iteration 301 / 2770) loss: 1.68315
(iteration 401 / 2770) loss: 1.652731
(Epoch 2 / 10) train acc: 0.472000; val_acc: 0.480000
(iteration 501 / 2770) loss: 1.681828
(iteration 601 / 2770) loss: 1.552158
(Epoch 3 / 10) train acc: 0.510000; val_acc: 0.465000
(iteration 701 / 2770) loss: 1.631960
(iteration 801 / 2770) loss: 1.484787
(Epoch 4 / 10) train acc: 0.506000; val_acc: 0.471000
(iteration 901 / 2770) loss: 1.538187
(iteration 1001 / 2770) loss: 1.486124
(iteration 1101 / 2770) loss: 1.434921
(Epoch 5 / 10) train acc: 0.543000; val_acc: 0.511000
(iteration 1201 / 2770) loss: 1.356131
(iteration 1301 / 2770) loss: 1.462831
(Epoch 6 / 10) train acc: 0.547000; val_acc: 0.513000
(iteration 1401 / 2770) loss: 1.301775
(iteration 1501 / 2770) loss: 1.302079
(Epoch 7 / 10) train acc: 0.553000; val_acc: 0.497000
(iteration 1601 / 2770) loss: 1.366449
(iteration 1701 / 2770) loss: 1.385570
(iteration 1801 / 2770) loss: 1.342451
(Epoch 8 / 10) train acc: 0.563000; val_acc: 0.529000
(iteration 1901 / 2770) loss: 1.313338
(iteration 2001 / 2770) loss: 1.538187
(Epoch 9 / 10) train acc: 0.562000; val_acc: 0.521000
(iteration 2101 / 2770) loss: 1.400773
(iteration 2201 / 2770) loss: 1.350052
(Epoch 10 / 10) train acc: 0.609000; val_acc: 0.540000
```

```
In [11]: # Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k-')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



Multilayer Neural Network

Now, we implement a multi-layer neural network.

Read through the `FullyConnectedNet` class in the file `nnutils/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. There will be lines for batchnorm and dropout layers and caches; ignore these all for now. That'll be in HW #4.

```
In [49]: N, D, H1, H2, C = 2, 15, 20, 30, 10
x = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
    print('Running check with reg = {}'.format(reg))
    model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                              reg=reg, weight_scale=5e-2, dtype=np.float64)

    loss, grads = model.loss(x, y)
    print('Initial loss: {}'.format(loss))

    f = lambda: model.loss(x, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('({}) relative error: {}'.format(name, rel_error(grad_num, grads[name])))

Running check with reg = 0
Initial loss: 2.303869879794544
W1 relative error: 1.289701627275205e-06
W2 relative error: 2.812817056147554e-06
b1 relative error: 1.42404896128542e-06
b2 relative error: 1.72889405818046e-09
b3 relative error: 1.210598878662188e-10
Running check with reg = 3.14
Initial loss: 7.20631810395218
W1 relative error: 1.3895236383682937e-08
W2 relative error: 1.6927821419208787e-08
b3 relative error: 1.42404896128542e-06
b1 relative error: 3.103688355030958e-08
b2 relative error: 1.52959204602250e-09
b3 relative error: 1.6222947517855002e-10
```

```
In [50]: # Use the three layer neural network to overfit a small dataset.

num_train = 50
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

#### !!!!!
# Play around with the weight_scale and learning_rate so that you can overfit a small dataset.
# It should be 0.8 or 0.9 if it relates to dout through multiplication with w, which is N x D after reshaping
# db should be M if it is just the sum over dout examples

weight_scale = 2e-2
learning_rate = 3e-3

model = FullyConnectedNet([100, 100],
                          weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                    'learning_rate': learning_rate,
                },
                solver=train())

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```



layers.py

```
In [51]: import numpy as np
import pdb

def affine_forward(x, w, b):
    """
    Computes the forward pass for an affine (fully-connected) layer.

    The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
    examples, where each example x[i] has shape (d_1, ..., d_k). We will
    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
    then transform it to an output vector of dimension M.

    Inputs:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """
    # YOUR CODE HERE:
    # - Calculate the output of the forward pass. Notice the dimensions
    # - of w are D x M, which is the transpose of what we did in earlier
    # assignments.
    # =====
    x_trans = x.reshape(x.shape[0], -1)
    out = np.dot(x_trans, w) + b

    # =====
    # END YOUR CODE HERE

    cache = (x, w, b)
    return out, cache

def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: tuple of:
      - x: Input data, of shape (N, d_1, ..., d_k)
      - w: Weights, of shape (D, M)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    # YOUR CODE HERE:
    # - Calculate the gradients for the backward pass.
    # =====
    # dout is N x M
    # dx should be N x d_1 x ... x d_k; it relates to dout through multiplication with w, which is D x M
    # dw should be D x M; it relates to dout through multiplication with x, which is N x D after reshaping
    # db should be M if it is just the sum over dout examples
    x_trans = x.reshape(x.shape[0], -1)
    dx = np.dot(dout, w.T)
    dx = dx.reshape(x.shape)
    dw = np.dot(x_trans, dout)
    db = np.sum(dout, axis=0)

    # =====
    # END YOUR CODE HERE

    return dx, dw, db

def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Inputs:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # YOUR CODE HERE:
    # - Implement the ReLU forward pass.
    # =====
    f = lambda x: x * (x > 0)
    out = f(x)

    # =====
    # END YOUR CODE HERE

    cache = x
    return out, cache

def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLUs).

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    # YOUR CODE HERE:
    # - Implement the ReLU backward pass.
    # =====
    f = lambda x: x * (x > 0)
    out = f(x)

    # =====
    # END YOUR CODE HERE

    cache = x
    return out, cache

def softmax_loss(x, y):
    """
    Computes the loss and gradient using multiclass SVM classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
      for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: scalar giving the loss
    - dx: Gradient of the loss with respect to x

    probs = np.exp(x - np.max(x, axis=1, keepdims=True))
    probs /= np.sum(probs, axis=1, keepdims=True)
    loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx

def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
      for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: scalar giving the loss
    - dx: Gradient of the loss with respect to x

    probs = np.exp(x - np.max(x, axis=1, keepdims=True))
    probs /= np.sum(probs, axis=1, keepdims=True)
    loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```

neural_net.py


```

import numpy as np
import matplotlib.pyplot as plt

class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network. The net has an input dimension of
    N, a hidden layer dimension of H, and performs classification over C classes.
    We train the network with a softmax loss function and L2 regularization on the
    weight matrices. The network uses a ReLU nonlinearity after the first fully
    connected layer.

    In other words, the network has the following architecture:

    input - fully connected layer - ReLU - fully connected layer - softmax

    The outputs of the second fully-connected layer are the scores for each class.
    """

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        """
        Initialize the model. Weights are initialized to small random values and
        biases are initialized to zero. Weights and biases are stored in the
        variable self.params, which is a dictionary with the following keys:

        W1: First layer weights; has shape (H, D)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (C, H)
        b2: Second layer biases; has shape (C,)

        Inputs:
        - input_size: The dimension D of the input data.
        - hidden_size: The number of neurons H in the hidden layer.
        - output_size: The number of classes C.
        """
        self.params = {}
        self.params['W1'] = std * np.random.randn(hidden_size, input_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = std * np.random.randn(output_size, hidden_size)
        self.params['b2'] = np.zeros(output_size)

    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural
        network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
          an integer in the range 0 <= y[i] < C. This parameter is optional; if it
          is not passed then we only return scores, and if it is passed then we
          instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, return a tuple of:
        - loss: loss (data loss and regularization loss) for this batch of training
          samples.
        - grads: Dictionary mapping parameter names to gradients of those parameters
          with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

        # Compute the forward pass
        scores = None

        # YOUR CODE HERE:
        # 1. Calculate the output scores of the neural network. This includes the
        #    softmax loss and the L2 regularization for W1 and W2. Store the
        #    total loss in the variable 'loss'. Multiply the regularization
        #    loss by 0.5 (in addition to the factor reg).
        # 2. Compute the backward pass. Compute the derivatives of the
        #    weights and the biases. Store the results in the grads
        #    dictionary. e.g., grads['W1'] should store the gradient for
        #    W1, and be of the same size as W1.

        update_scores = class_probabilities
        update_scores(np.arange(N), y)
        update_scores /= N

        grads['W2'] = np.dot(H1.T, update_scores).T
        grads['b2'] = np.sum(update_scores, axis=0)
        dH2 = np.dot(update_scores, W2)

        dLdA = dH2
        dLdA[H1 <= 0] = 0

        grads['W1'] = np.dot(dLdA.T, X)
        grads['b1'] = np.sum(dLdA, axis=0, keepdims=True)

        grads['W2'] += reg * W2
        grads['W1'] += reg * W1

        # END YOUR CODE HERE

        return loss, grads

    def train(self, X, y, X_val, y_val,
              learning_rate=1e-3, learning_rate_decay=0.95,
              reg=5e-5, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this neural network using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) giving training data.
        - y: A numpy array of shape (N,) giving training labels; y[i] = c means that
          X[i] has label c, where 0 <= c < C.
        - X_val: A numpy array of shape (N_val, D) giving validation data.
        - y_val: A numpy array of shape (N_val,) giving validation labels.
        - learning_rate: Scalar giving learning rate for optimization.
        - learning_rate_decay: Scalar giving factor used to decay the learning rate
          after each epoch.
        - reg: Scalar giving regularization strength.
        - num_iters: Number of steps to take when optimizing.
        - batch_size: Number of training examples to use per step.
        - verbose: boolean; if true print progress during optimization.
        """
        num_train = X.shape[0]
        iterations_per_epoch = max(num_train / batch_size, 1)

        # Use SGD to optimize the parameters in self.model
        loss_history = []
        train_acc_history = []
        val_acc_history = []

        for it in np.arange(num_iters):
            X_batch = None
            y_batch = None

            # YOUR CODE HERE:
            # Create a minibatch by sampling batch_size samples randomly.

            random_indices = np.random.choice(num_train, batch_size)
            X_batch = X[random_indices]
            y_batch = y[random_indices]

            # Compute loss and gradients using the current minibatch
            loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
            loss_history.append(loss)

            # Perform a gradient descent step using the minibatch to update
            # all parameters (i.e., W1, W2, b1, and b2).

            self.params['W2'] -= learning_rate * grads['W2']
            self.params['W1'] -= learning_rate * grads['W1']
            self.params['b2'] -= learning_rate * grads['b2']
            self.params['b1'] -= learning_rate * grads['b1']

            # END YOUR CODE HERE

            if verbose and it % 100 == 0:
                print('iteration %i / %i: loss %f' % (it, num_iters, loss))

            # Every epoch, check train and val accuracy and decay learning rate.
            if it % iterations_per_epoch == 0:
                # Check accuracy
                train_acc = (self.predict(X_batch) == y_batch).mean()
                val_acc = (self.predict(X_val) == y_val).mean()
                train_acc_history.append(train_acc)
                val_acc_history.append(val_acc)

                # Decay learning rate
                learning_rate *= learning_rate_decay

            return {
                'loss_history': loss_history,
                'train_acc_history': train_acc_history,
                'val_acc_history': val_acc_history,
            }

    def predict(self, X):
        """
        Use the trained weights of this two-layer network to predict labels for
        data points. For each data point we predict scores for each of the C
        classes, and assign each data point to the class with the highest score.

        Inputs:
        - X: A numpy array of shape (N, D) giving N D-dimensional data points to
          classify.

        Returns:
        - y_pred: A numpy array of shape (N,) giving predicted labels for each
          of the elements of X. For all i, y_pred[i] = c means that X[i] is
          predicted to have class c, where 0 <= c < C.
        """
        y_pred = None

        # YOUR CODE HERE:
        # Predict the class given the input data.

        num_examples = X.shape[0]
        y_pred = np.empty((num_examples, ), dtype=int)
        H1_preact = np.dot(X, self.params['W1'].T) + self.params['b1']

        # Apply ReLU
        H1 = np.maximum(0, H1_preact)

        # Second layer
        H2 = np.dot(H1, self.params['W2'].T) + self.params['b2']

        # Apply softmax
        softmax = np.exp(H2) / np.sum(np.exp(H2), axis=1, keepdims=True)

        for i in range(num_examples):
            max_index = np.argmax(softmax[i])
            y_pred[i] = max_index

        # END YOUR CODE HERE

        return y_pred

```

fc_net.py

In [3]:

```

import numpy as np

from layers import *
from layer_utils import *

class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network with ReLU nonlinearity and
    softmax loss that uses a modular layer design. We assume an input dimension
    of D, a hidden dimension of H, and perform classification over C classes.

    The architecture should be affine - relu - affine - softmax.

    Note that this class does not implement gradient descent; instead, it
    will interact with a separate Solver object that is responsible for running
    optimization.

    The learnable parameters of the model are stored in the dictionary
    self.params that maps parameter names to numpy arrays.
    """

    def __init__(self, input_dim=3*32*32, hidden_dim=100, num_classes=10,
                 dropout=0, weight_scale=1e-3, reg=0.0):
        """
        Initialize a new network.

        Inputs:
        - input_dim: An integer giving the size of the input
        - hidden_dim: An integer giving the size of the hidden layer
        - num_classes: An integer giving the number of classes to classify
        - dropout: Scalar between 0 and 1 giving dropout strength.
        - weight_scale: Scalar giving the standard deviation for random
          initialization of the weights.
        - reg: Scalar giving L2 regularization strength.
        """
        self.params = {}
        self.reg = reg

        # YOUR CODE HERE:
        # Initialize W1, W2, b1, and b2. Store these as self.params['W1'],
        # self.params['W2'], self.params['b1'] and self.params['b2']. The
        # biases are initialized to zero and the weights are initialized
        # so that each parameter has mean 0 and standard deviation weight_scale.
        # The dimensions of W1 should be (input_dim, hidden_dim) and the
        # dimensions of W2 should be (hidden_dim, num_classes)

        W1_shape = (input_dim, hidden_dim)
        W2_shape = (hidden_dim, num_classes)

        self.params['W1'] = np.random.normal(loc=0.0, scale=weight_scale, size=W1_shape)
        self.params['b1'] = np.zeros(hidden_dim)
        self.params['W2'] = np.random.normal(loc=0.0, scale=weight_scale, size=W2_shape)
        self.params['b2'] = np.zeros(num_classes)

        # END YOUR CODE HERE

    def loss(self, X, y=None):
        """
        Compute loss and gradient for a minibatch of data.

        Inputs:
        - X: Array of input data of shape (N, d_1, ..., d_k)
        - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

        Returns:
        If y is None, then run a test-time forward pass of the model and return:
        - scores: Array of shape (N, C) giving classification scores, where
          scores[i, c] is the classification score for X[i] and class c.

        If y is not None, then run a training-time forward and backward pass and
        return a tuple of:
        - loss: Scalar value giving the loss
        - grads: Dictionary with the same keys as self.params, mapping parameter
          names to gradients of the loss with respect to those parameters.
        """
        scores = None

        # YOUR CODE HERE:
        # Implement the forward pass of the two-layer neural network. Store
        # the class scores as the variable 'scores'. Be sure to use the layers
        # you prior implemented.

        W1 = self.params['W1']
        b1 = self.params['b1']
        W2 = self.params['W2']
        b2 = self.params['b2']

        H, cache_h = affine_relu_forward(X, W1, b1)
        Z, cache_z = affine_forward(H, W2, b2)

        scores = Z

        # END YOUR CODE HERE

        # If y is None then we are in test mode so just return scores
        if y is None:
            return scores

        loss, grads = 0, {}

        # YOUR CODE HERE:
        # Implement the backward pass of the two-layer neural net. Store
        # the loss as the variable 'loss' and store the gradients in the
        # 'grads' dictionary. For the grads dictionary, grads['W1'] holds
        # the gradient for W1, grads['b1'] holds the gradient for b1, etc.
        # grads[k] holds the gradient for self.params[k].
        # Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
        # for each W. Be sure to include the 0.5 multiplying factor to
        # match our implementation.
        # And be sure to use the layers you prior implemented.

        loss, dz = softmax_loss(scores, y)
        loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2))

        dx, dw1, db1 = affine_backward(dz, cache_h)
        dh, dw2, db2 = affine_backward(dz, cache_z)

        grads['W1'] = dw1 + self.reg * W1
        grads['b1'] = db1
        grads['W2'] = dw2 + self.reg * W2
        grads['b2'] = db2

        # END YOUR CODE HERE

        return loss, grads

class FullyConnectedNet(object):
    """
    A fully-connected neural network with an arbitrary number of hidden layers,
    ReLU nonlinearities, and a softmax loss function. This will also implement
    dropout and batch normalization as options. For a network with L layers,
    the architecture will be

    (affine - [batch norm] - relu - [dropout]) x (L - 1) - affine - softmax

    where batch normalization and dropout are optional, and the (...) block is
    repeated L - 1 times.

    Similar to the TwoLayerNet above, learned parameters are stored in the
    self.params dictionary and will be learned using the Solver class.
    """

    def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
                 dropout=0, use_batchnorm=False, reg=0.0,
                 weight_scale=1e-2, dtype=np.float32, seed=None):
        """
        Initialize a new FullyConnectedNet.

        Inputs:
        - hidden_dims: A list of integers giving the size of each hidden layer.
        - input_dim: An integer giving the size of the input.
        - num_classes: An integer giving the number of classes to classify.
        - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
          the network should not use dropout at all.
        - use_batchnorm: Whether or not the network should use batch normalization.
        - reg: Scalar giving L2 regularization strength.
        - weight_scale: Scalar giving the standard deviation for random
          initialization of the weights.
        - dtype: A numpy datatype object; all computations will be performed using
          this datatype. float32 is faster but less accurate, so you should use
          float64 for numeric gradient checking.
        - seed: If not None, then pass this random seed to the dropout layers. This
          will make the dropout layers deterministic so we can gradient check the
          model.
        """
        self.use_batchnorm = use_batchnorm
        self.use_dropout = dropout > 0
        self.reg = reg
        self.num_layers = 1 + len(hidden_dims)
        self.dtype = dtype
        self.params = {}

        # YOUR CODE HERE:
        # Initialize all parameters of the network in the self.params dictionary.
        # The weights and biases of layer i are W_i and b_i, and in general the
        # weights are initialized to zero and the weights are initialized
        # so that each parameter has mean 0 and standard deviation weight_scale.
        # For i in np.arange(1, self.num_layers+1):
        name_W = 'W'+str(i)
        name_b = 'b'+str(i)

        if i == 1: # first layer
            self.params[name_W] = np.random.normal(loc=0.0, scale=weight_scale, size=(input_dim, hidden_dims[i-1]))
            self.params[name_b] = np.zeros(hidden_dims[i-1])
        elif i == self.num_layers: # last layer
            self.params[name_W] = np.random.normal(loc=0.0, scale=weight_scale, size=(hidden_dims[i-2], num_classes))
            self.params[name_b] = np.zeros(num_classes)
        else: # intermediate layers
            self.params[name_W] = np.random.normal(loc=0.0, scale=weight_scale, size=(hidden_dims[i-2], hidden_dims[i-1]))
            self.params[name_b] = np.zeros(hidden_dims[i-1])

        # END YOUR CODE HERE

        # When using dropout we need to pass a dropout_param dictionary to each
        # dropout layer so that the layer knows the dropout probability and the mode
        # (train / test). You can pass the same dropout_param to each dropout layer.
        self.dropout_param = {}
        if self.use_dropout:
            self.dropout_param = {'mode': 'train', 'p': dropout}
            if seed is not None:
                self.dropout_param['seed'] = seed

        # With batch normalization we need to keep track of running means and
        # variances, so we need to pass a special bn_param object to each batch
        # normalization layer. You should pass self.bn_params[0] to the forward pass
        # of the first batch normalization layer, self.bn_params[1] to the forward
        # pass of the second batch normalization layer, etc.
        self.bn_params = []
        if self.use_batchnorm:
            self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]

        # Cast all parameters to the correct datatype
        for k, v in self.params.items():
            self.params[k] = v.astype(dtype)

    def loss(self, X, y=None):
        """
        Compute loss and gradient for the fully-connected net.

        Input / output: Same as TwoLayerNet above.
        """
        X = X.astype(self.dtype)
        mode = 'test' if y is None else 'train'

        # Set train/test mode for batchnorm params and dropout param since they
        # behave differently during training and testing.
        if self.dropout_param is not None:
            self.dropout_param['mode'] = mode
        if self.use_batchnorm:
            for bn_param in self.bn_params:
                bn_param['mode'] = mode

        scores = None

        # YOUR CODE HERE:
        # Implement the forward pass of the FC net and store the output
        # scores as the variable 'scores'.

        N = []
        cache_h = []
        for i in np.arange(1, self.num_layers + 1):
            name_W = 'W'+str(i)
            name_b = 'b'+str(i)

            if i == 1:
                H = affine_relu_forward(X, self.params[name_W], self.params[name_b])[0]
                cache_h.append(affine_relu_forward(X, self.params[name_W], self.params[name_b])[1])
            elif i == self.num_layers:
                scores = affine_forward(H[i-2], self.params[name_W], self.params[name_b])[0]
                cache_h.append(affine_forward(H[i-2], self.params[name_W], self.params[name_b])[1])
            else:
                H = affine_relu_forward(H[i-2], self.params[name_W], self.params[name_b])[0]
                cache_h.append(affine_relu_forward(H[i-2], self.params[name_W], self.params[name_b])[1])

        # END YOUR CODE HERE

        # If test mode return early
        if mode == 'test':
            return scores

        loss, grads = 0, {}

        # YOUR CODE HERE:
        # Implement the backwards pass of the FC net and store the gradients
        # in the grads dict, so that grads[k] is the gradient of self.params[k]
        # with respect to the loss. Be sure your L2 regularization includes a 0.5 factor.

        loss, dz = softmax_loss(scores, y)
        dh = []
        for i in np.arange(self.num_layers, 0, -1):
            name_W = 'W'+str(i)
            name_b = 'b'+str(i)
            loss += (0.5 * self.reg * np.sum(self.params[name_W]**2 + self.params[name_W]**2))
            if i == self.num_layers:
                dh1, grads[name_W], grads[name_b] = affine_backward(dz, cache_h[self.num_layers-1])
                dh.append(dh1)
            else:
                dh1, grads[name_W], grads[name_b] = affine_relu_backward(dh1, cache_h[i-1])
                dh.append(dh1)

            grads[name_W] += self.reg * self.params[name_W]

        # END YOUR CODE HERE

        return loss, grads

```

In [3]: