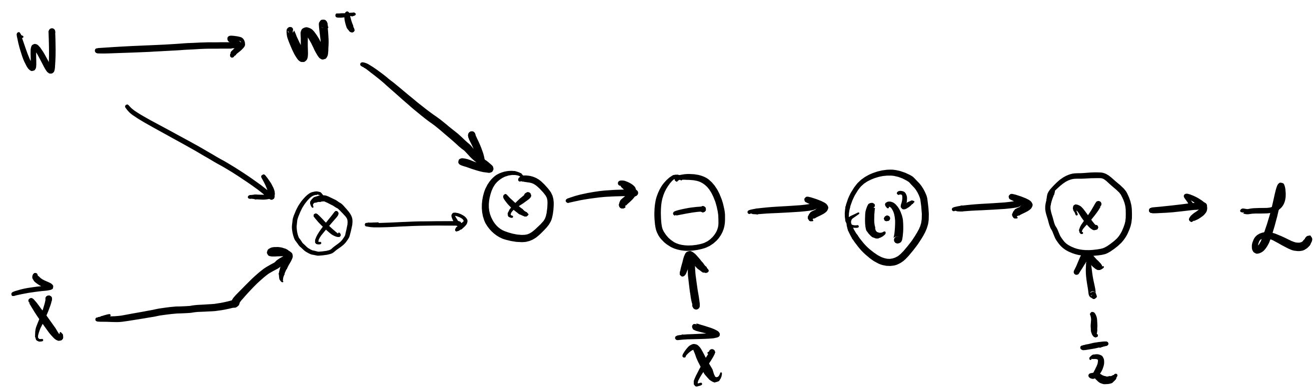
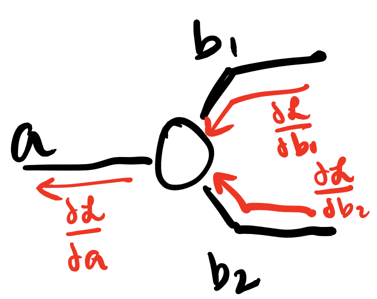
1. Backpropagation for auto encoders

a) The dimensionality reduction key to an autoencoder comes from $\overset{m\times n}{W}\overset{n\times 1}{\vec{x}}$ which is $m\times 1$ and has lower dimensionality than $\vec{x}$ $(n\times 1)$ since $m<n$. However, multiplication by W^T is the decoding step that reconstructs data from the encodings since W^T is $n\times m$ so $\overset{n\times m}{W^T}\overset{m\times n}{W}\overset{n\times 1}{\vec{x}}$ is $n\times 1$, the same size as input $\vec{x}$, therefore preserving dimensionality.

Autoencoders, especially linear ones like in this example, are similar to PCA in which we find the eigenvectors and eigenvalues of the covariance matrix that minimizes the reconstruction error. Minimizing the given loss function achieves the same objective of finding W that saves the important information about $\vec{x}$.

b) 

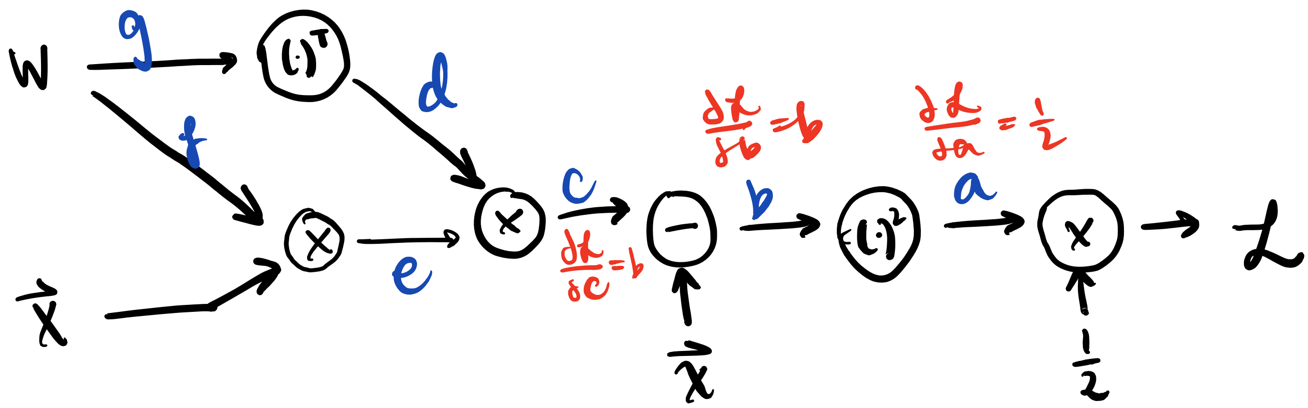
c) As per the law of total derivatives, we sum over the partial derivatives of each of the two paths in order to count each path's contribution to $\nabla_W\mathcal{L}$. For example, in the shown computational graph, $\nabla_a\mathcal{L}$ is:



$$\frac{\partial\mathcal{L}}{\partial a} = \sum_{i=1}^{n} \frac{\partial\mathcal{L}}{\partial b_i}\cdot\frac{\partial b_i}{\partial a} = \frac{\partial b_1}{\partial a}\frac{\partial\mathcal{L}}{\partial b_1} + \frac{\partial b_2}{\partial a}\frac{\partial\mathcal{L}}{\partial b_2}$$

so both $\frac{\partial\mathcal{L}}{\partial b_1}$ and $\frac{\partial\mathcal{L}}{\partial b_2}$ influence the value of $\frac{\partial\mathcal{L}}{\partial a}$.

d) $\mathcal{L} = \frac{1}{2}\|W^T W \vec{x} - \vec{x}\|^2$



$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial}{\partial a}\left(\frac{1}{2}a\right) = \frac{1}{2}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial a}{\partial b}\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial}{\partial b}b^2 \cdot \frac{1}{2} = 2b \cdot \frac{1}{2} = b$$

$$\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial b}{\partial c}\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial c}(c - \vec{x}) \cdot b = 1 \cdot b = b$$

$$\frac{\partial \mathcal{L}}{\partial d} = \frac{\partial c}{\partial d}\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial (W^T W x)}{\partial (W^T)} \cdot b = \underset{1\times m}{(Wx)^T}\ \underset{n\times 1}{b} = b(Wx)^T$$

$$\frac{\partial \mathcal{L}}{\partial e} = \frac{\partial c}{\partial e}\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial (W^T W x)}{\partial (Wx)} \cdot b = Wb$$

$$\frac{\partial \mathcal{L}}{\partial f} = \frac{\partial e}{\partial f}\frac{\partial \mathcal{L}}{\partial e} = \frac{\partial (Wx)}{\partial W} \cdot Wb = \underset{1\times n}{x^T}\ \underset{m\times 1}{Wb} \rightarrow Wbx^T$$

$$\frac{\partial \mathcal{L}}{\partial g} = \left(\frac{\partial \mathcal{L}}{\partial d}\right)^T = Wxb^T$$

$$\frac{\partial \mathcal{L}}{\partial W} = \cancel{\frac{\partial f}{\partial W}}^{1} \cdot \frac{\partial \mathcal{L}}{\partial f} + \cancel{\frac{\partial g}{\partial W}}^{1} \frac{\partial \mathcal{L}}{\partial g}$$

$$= Wbx^T + Wxb^T$$

$$= \boxed{W(W^T W x - x)x^T + Wx(W^T W x - x)^T}$$

two_layer_nn

February 2, 2021

0.1 This is the 2-layer neural network workbook for ECE 247 Assignment #3

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

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

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

[94]:
```
import random
import numpy as np
from cs231n.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
```

0.2 Toy example

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

[95]:
```
from nndl.neural_net import TwoLayerNet
```

[96]:
```
# 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
```

```python
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

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

0.2.1 Compute forward pass scores

[98]:
```python
## 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:
3.381231233889892e-08
```

0.2.2 Forward pass loss

[107]:
```python
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

```
X: (5, 4)
W1: (10, 4)
b1: (10,)
W2: (3, 10)
b2: (3,)
grad: [[ 0.04887766 -0.23243768  0.02662056 -0.06621344 -0.31432313  0.19727518
  -0.54603789 -0.17899805  0.19214611 -0.28456731]
 [-0.115572    0.35072881 -0.06022936  0.03328376  0.56945447  0.38288047
   0.31246909  0.12811425 -0.36300479  0.22308082]
 [ 0.06669435 -0.11829113  0.0336088   0.03292968 -0.25513134 -0.58015565
   0.2335688   0.0508838   0.17085868  0.06148649]]
Difference between your loss and correct loss:
0.0
```

[100]:
```python
print(loss)
```

```
1.071696123862817
```

0.2.3 Backward pass

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

[155]:
```python
from cs231n.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 less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
↪verbose=False)
    print('{} max relative error: {}'.format(param_name,␣
↪rel_error(param_grad_num, grads[param_name])))
```

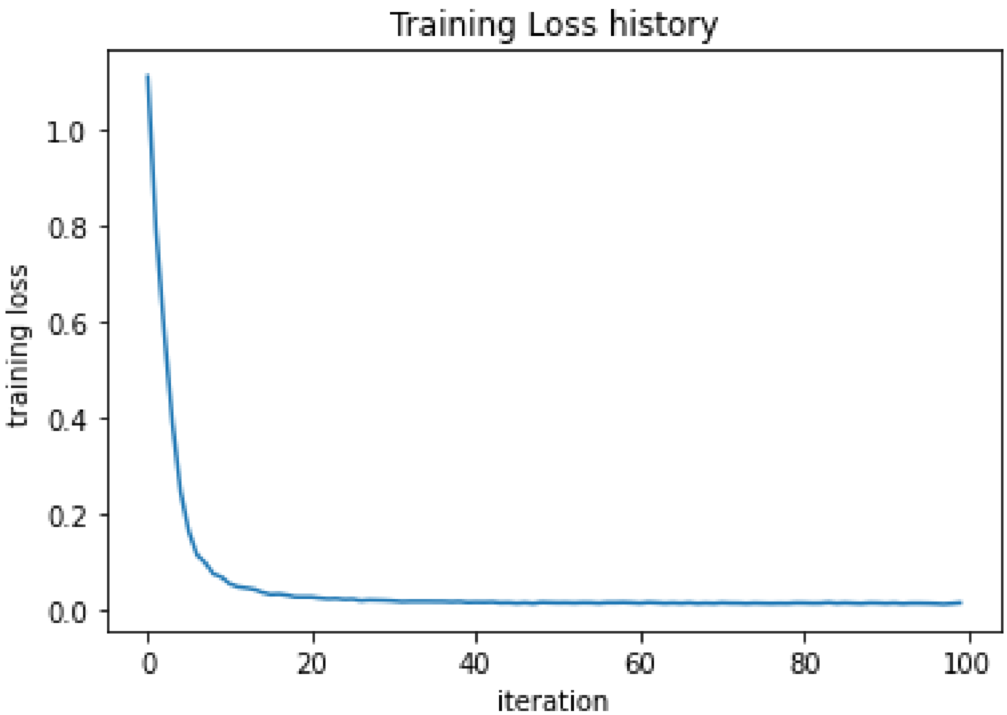
```
b2 max relative error: 1.8391748601536041e-10
W2 max relative error: 2.9632227682005116e-10
b1 max relative error: 3.1726806716844575e-09
W1 max relative error: 1.2832874456864775e-09
```

0.2.4 Training the network

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

[160]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, 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:  0.014497864587765997
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[162]:
```
from cs231n.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. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-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))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
```

```
    X_test = X_test[mask]
    y_test = y_test[mask]

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


# Invoke the above function to get our data.
X_train, y_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,)
```

0.3.1 Running SGD

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

[167]:
```
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=1000, batch_size=200,
            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 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.9901688623083942
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

0.4 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.

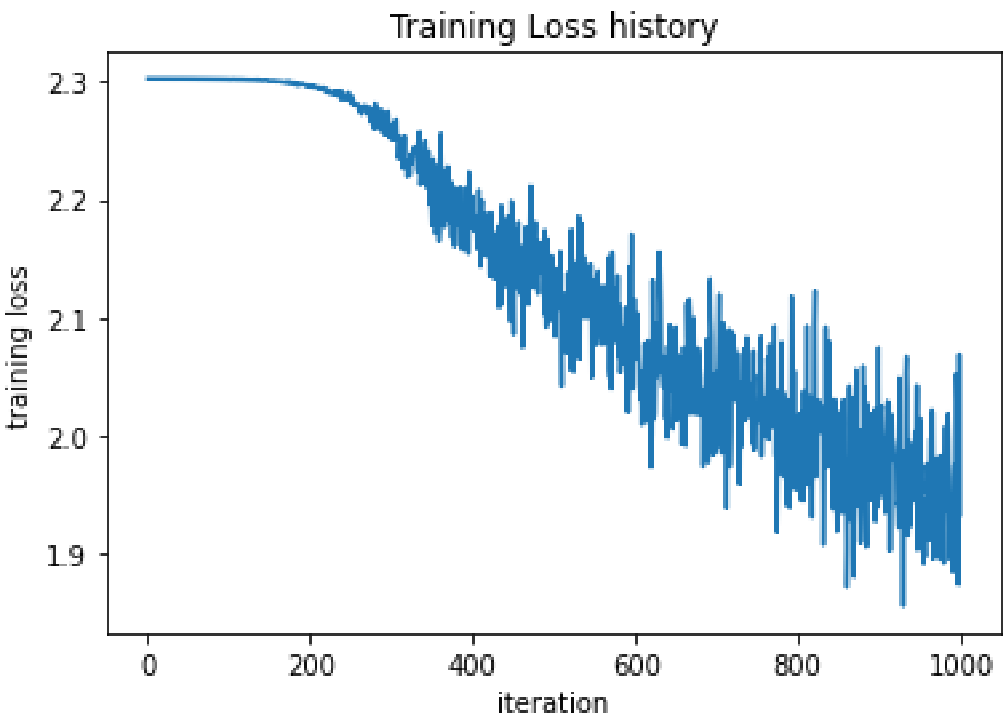
(2) How should you fix the problems you identified in (1)?

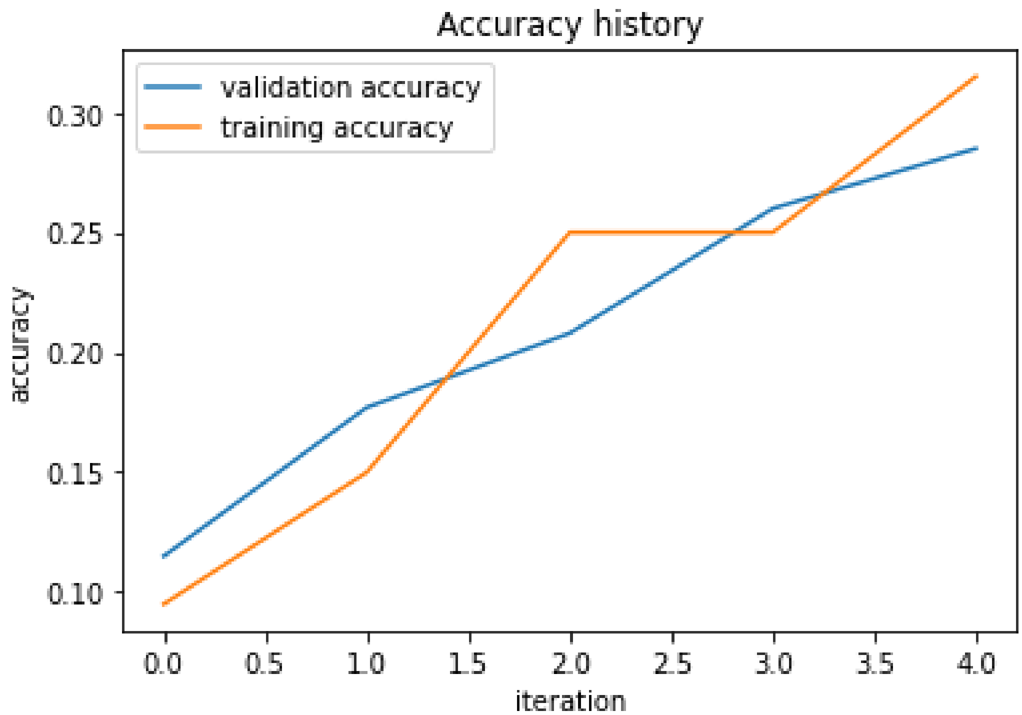
[168]:
```
stats['train_acc_history']
```

[168]: [0.095, 0.15, 0.25, 0.25, 0.315]

[171]:
```
# ================================================================ #
# 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

# Graph: Loss vs. Iteration
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()

# Graph: Training accuracy and Validation accuracy vs. Iteration
plt.plot(stats['val_acc_history'], label='validation accuracy')
```

```
plt.plot(stats['train_acc_history'], label='training accuracy')
plt.xlabel('iteration')
plt.ylabel('accuracy')
plt.title('Accuracy history')
plt.legend()
plt.show()

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





0.5 Answers:

(1) The training accuracy is not the best perhaps because the model needs to train for a greater number of iterations. Since the training and validation accuracy are still increasing while loss is still decreasing, there are no signs of overtraining and the model is expected to improve with further training.

(2) This can be improved by training the model for a greater number of iterations, and/or increasing the learning rate. The fairly linear training loss indicates low learning rate, where as good learning rate should yield a loss curve that is more exponential, decaying faster to a low value before plateauing. In addition, the noisy cost function indicates that the batch size may be a little too low.

0.6 Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

[179]:
```
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.
```

```python
#   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((X - 28%)) / %22, 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)!
# ================================================================ #

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=10000, batch_size=200,
            learning_rate=2e-3, 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)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
best_net = net
```

```
iteration 0 / 10000: loss 2.30277699762532
iteration 100 / 10000: loss 1.7590223097586706
iteration 200 / 10000: loss 1.649073065982276
iteration 300 / 10000: loss 1.6762319172480356
iteration 400 / 10000: loss 1.612993323184264
iteration 500 / 10000: loss 1.6210144723420281
iteration 600 / 10000: loss 1.514070254883771
iteration 700 / 10000: loss 1.4774067330033271
iteration 800 / 10000: loss 1.5440622181493455
iteration 900 / 10000: loss 1.4779415541983278
iteration 1000 / 10000: loss 1.4416514497443955
iteration 1100 / 10000: loss 1.577367714423431
iteration 1200 / 10000: loss 1.4968606973796912
iteration 1300 / 10000: loss 1.319823653825999
iteration 1400 / 10000: loss 1.3773639178158408
iteration 1500 / 10000: loss 1.4841558005024664
iteration 1600 / 10000: loss 1.445227779767124
iteration 1700 / 10000: loss 1.5136419395755094
```

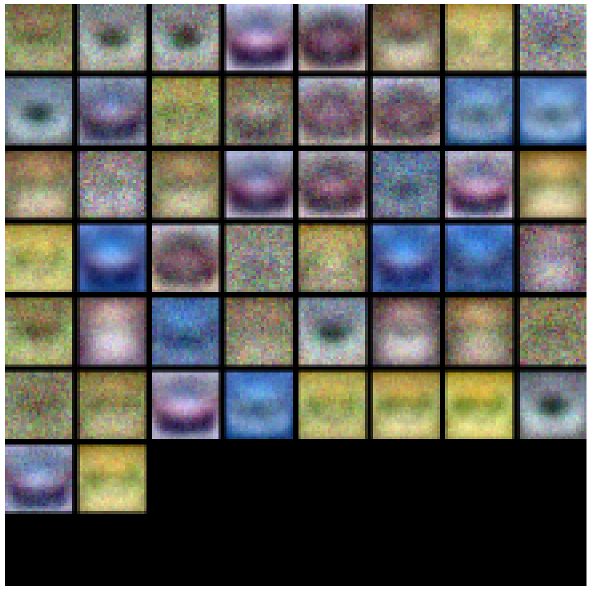
```
iteration 1800 / 10000: loss 1.3263004695031
iteration 1900 / 10000: loss 1.3872988353033355
iteration 2000 / 10000: loss 1.2927478296945654
iteration 2100 / 10000: loss 1.4682625196140446
iteration 2200 / 10000: loss 1.335705675812533
iteration 2300 / 10000: loss 1.47526280415969
iteration 2400 / 10000: loss 1.3911241612656953
iteration 2500 / 10000: loss 1.464347856757441
iteration 2600 / 10000: loss 1.4153576570463686
iteration 2700 / 10000: loss 1.554321208908832
iteration 2800 / 10000: loss 1.3950331560188178
iteration 2900 / 10000: loss 1.4697978964435408
iteration 3000 / 10000: loss 1.4444495889109474
iteration 3100 / 10000: loss 1.2645820714493545
iteration 3200 / 10000: loss 1.4129759267640902
iteration 3300 / 10000: loss 1.3806700811002401
iteration 3400 / 10000: loss 1.27042886032785
iteration 3500 / 10000: loss 1.3909187580224094
iteration 3600 / 10000: loss 1.431084459547198
iteration 3700 / 10000: loss 1.3705905442717101
iteration 3800 / 10000: loss 1.3343697350445665
iteration 3900 / 10000: loss 1.3324006360153735
iteration 4000 / 10000: loss 1.281674707223195
iteration 4100 / 10000: loss 1.44969319689627
iteration 4200 / 10000: loss 1.317491392163756
iteration 4300 / 10000: loss 1.430566466171759
iteration 4400 / 10000: loss 1.2225556828935418
iteration 4500 / 10000: loss 1.377085371850397
iteration 4600 / 10000: loss 1.3948210972668553
iteration 4700 / 10000: loss 1.2362217195165806
iteration 4800 / 10000: loss 1.1735702582717047
iteration 4900 / 10000: loss 1.2383164001263367
iteration 5000 / 10000: loss 1.2212496465152347
iteration 5100 / 10000: loss 1.3336080898810336
iteration 5200 / 10000: loss 1.380522098574305
iteration 5300 / 10000: loss 1.3225571351429717
iteration 5400 / 10000: loss 1.290874976528839
iteration 5500 / 10000: loss 1.3448199268461096
iteration 5600 / 10000: loss 1.353194463520747
iteration 5700 / 10000: loss 1.2530415431616946
iteration 5800 / 10000: loss 1.550389141255752
iteration 5900 / 10000: loss 1.2472179118500868
iteration 6000 / 10000: loss 1.134106014172791
iteration 6100 / 10000: loss 1.2308295168300458
iteration 6200 / 10000: loss 1.215778640447147
iteration 6300 / 10000: loss 1.3452116283330344
iteration 6400 / 10000: loss 1.2251961947024959
iteration 6500 / 10000: loss 1.3921535345623532
```

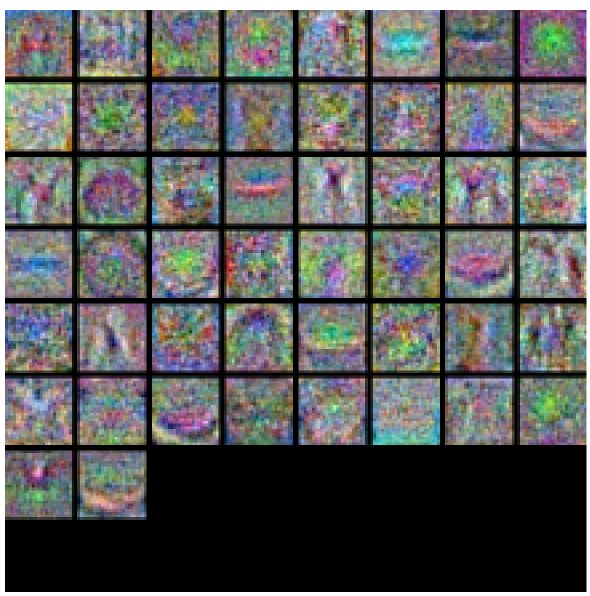
```
iteration 6600 / 10000: loss 1.2290011627007436
iteration 6700 / 10000: loss 1.308043064033643
iteration 6800 / 10000: loss 1.2509223405090142
iteration 6900 / 10000: loss 1.2568525135380415
iteration 7000 / 10000: loss 1.182291898238407
iteration 7100 / 10000: loss 1.324272759152664
iteration 7200 / 10000: loss 1.2488528774462804
iteration 7300 / 10000: loss 1.2586482307085
iteration 7400 / 10000: loss 1.1742444907591183
iteration 7500 / 10000: loss 1.3680070509586266
iteration 7600 / 10000: loss 1.2226660491542947
iteration 7700 / 10000: loss 1.246287617902513
iteration 7800 / 10000: loss 1.1968901360150193
iteration 7900 / 10000: loss 1.1783407800560042
iteration 8000 / 10000: loss 1.201826635513235
iteration 8100 / 10000: loss 1.1873109155915982
iteration 8200 / 10000: loss 1.1765934490056862
iteration 8300 / 10000: loss 1.2203564199359804
iteration 8400 / 10000: loss 1.2523691272291964
iteration 8500 / 10000: loss 1.24212834333846
iteration 8600 / 10000: loss 1.369467931851104
iteration 8700 / 10000: loss 1.2337298339262766
iteration 8800 / 10000: loss 1.275557484672944
iteration 8900 / 10000: loss 1.265332760338735
iteration 9000 / 10000: loss 1.2284385227907273
iteration 9100 / 10000: loss 1.2871236423924308
iteration 9200 / 10000: loss 1.2821524083750617
iteration 9300 / 10000: loss 1.2449503952294956
iteration 9400 / 10000: loss 1.2200834799427718
iteration 9500 / 10000: loss 1.181527947884218
iteration 9600 / 10000: loss 1.2436407481276635
iteration 9700 / 10000: loss 1.328991916919356
iteration 9800 / 10000: loss 1.1030879529903288
iteration 9900 / 10000: loss 1.1624475720555583
Validation accuracy:  0.511
```

[175]:

```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()
```

```
show_net_weights(subopt_net)
show_net_weights(best_net)
```





0.7 Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

0.8 Answer:

(1) The best net I arrived at has much more complex weights that capture image structure better than the suboptimal net.

0.9 Evaluate on test set

[177]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.515
```

0.10 neural_net.py

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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:
```

```python
    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 matrix scores of shape (N, C) where scores[i, c] is
    the score for class c on input X[i].

    If y is not 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.
    # ================================================================ #

    # result should be (5, 3) since N=5 for N training examples, C=3 for C␣
↪classes
    H1 = np.dot(X, W1.T) + b1 # (5,4)*(4,10) + (10,) = (5,10)
    H1 = np.maximum(np.zeros(H1.shape), H1) # ReLU(x) = max(0,x)
    scores = np.dot(H1, W2.T) + b2 # (5,10)*(10,3) + (3,) = (5,3)

    # ================================================================ #
    # 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 teh variable loss.  Multiply the regularization
        #         loss by 0.5 (in addition to the factor reg).
        # ================================================================ #

    # scores is num_examples by num_classes
    num_examples, num_classes = scores.shape

    e_A = np.exp(scores)
    summed_e_A = np.sum(e_A, axis=1) # sum across each of 5 rows -> sum per␣
↪class
    logged_summed_e_A = np.log(summed_e_A)
    A_y = scores[np.arange(num_examples), y] # scores of correct class␣
↪predictions
```

```
    regularization = 0.5 * reg * (np.sum(np.square(W1)) + np.sum(np.
↪square(W2))) # sum of squared values of weights
    loss = np.mean(logged_summed_e_A - A_y) + regularization

    # ================================================================ #
    # 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.
        # ================================================================ #

    # gradient of softmax without L2 regularization
    softmax_gradient = e_A / summed_e_A.reshape(num_examples, 1)
    softmax_gradient[np.arange(num_examples), y] -= 1
    softmax_gradient /= num_examples # (5,3)

    # gradients for L2 regularlization used in loss function
    gradient_reg_W2 = reg * W2 # d(0.5 * reg * (W1**2 + W2**2)) / d(W2) = reg *␣
↪W2
    gradient_reg_W1 = reg * W1 # d(0.5 * reg * (W1**2 + W2**2)) / d(W1) = reg *␣
↪W1

    grads['b2'] = np.sum(softmax_gradient.T, axis=1) # (3,5) -> sum across each␣
↪row/class -> (3,) <-> b2=(3,)

    gradient_1 = np.dot(H1.T, softmax_gradient).T # d(H1*W2.T) / d(W2.T)
    grads['W2'] = gradient_reg_W2 + gradient_1 # (3,10) <-> W2=(3,10)

    gradient_2 = np.dot(softmax_gradient, W2)
    gradient_2[H1 <= 0] = 0 # d(softmax(ReLu(W1.T*X + b1)*W2.T + b2)) / d(W1.
↪T*X + b1)

    grads['b1'] = np.sum(gradient_2, axis=0) # (10,) <-> b1=(10,)
    grads['W1'] = gradient_reg_W1 + np.dot(X.T, gradient_2).T # (10,4) <->␣
↪W1=(10,4)

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

```python
    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-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 f 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(int(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.
            # ================================================================ #

      idxs = np.random.choice(num_train, batch_size, replace=True) # replace␣
↪like in HW2?
      X_batch = X[idxs]
      y_batch = y[idxs]
```

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

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

    if verbose and it % 100 == 0:
      print('iteration {} / {}: loss {}'.format(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.
    # ================================================================ #

    H1 = np.dot(X, self.params['W1'].T) + self.params['b1'] # (5,4)*(4,10) +␣
↪(10,) = (5,10)
    H1 = np.maximum(np.zeros(H1.shape), H1) # ReLU(x) = max(0,x)
    scores = np.dot(H1, self.params['W2'].T) + self.params['b2'] #␣
↪(5,10)*(10,3) + (3,) = (5 examples, 3 classes)

    y_pred = np.argmax(scores, axis=1) # maximum value per row = class with␣
↪highest score

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

    return y_pred
```

FC_nets

February 2, 2021

1 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).

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, and their layer structure. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

1.1 Modular layers

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

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  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 derivative 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
```

[3]:

```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
↪eval_numerical_gradient_array
from cs231n.solver import Solver

import os
# alias kk os._exit(0)

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
↪autoreload-of-modules-in-ipython
%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))))
```

```
[4]: # 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,)
```

1.2 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 `nndl/layers.py` and the backward pass is `affine_backward`.

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

1.2.1 Affine layer forward pass

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

```
[5]: # Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

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

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),␣
 ↪output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# 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: 9.769849468192957e-10
```

1.2.2 Affine layer backward pass

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

[6]:
```python
# 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 w: affine_forward(x, w, b)[0], w,␣
 ↪dout)
db_num = eval_numerical_gradient_array(lambda b: 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: 2.42097799776016e-10
dw error: 1.3194027115746944e-10
db error: 7.944683283018363e-12
```

1.3 Activation layers

In this section you'll implement the ReLU activation.

1.3.1 ReLU forward pass

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

[7]:
```python
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
```

```
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  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: 4.999999798022158e-08
```

1.3.2 ReLU backward pass

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

[8]:
```
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: 3.2756219895652386e-12
```

1.4 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 `nndl/layer_utils.py`.

1.4.1 Affine-ReLU layers

We've implemented `affine_relu_forward()` and `affine_relu_backward` in `nndl/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.

[9]:
```
from nndl.layer_utils 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 w: affine_relu_forward(x, w,␣
↪b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: 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: 4.752217859141691e-10
dw error: 1.3417348910961532e-10
db error: 1.8928885995207832e-11
```

1.5 Softmax and SVM losses

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

[10]:
```
num_classes, num_inputs = 10, 50
x = 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: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print('Testing svm_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))

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('\nTesting softmax_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing svm_loss:
loss: 8.999722165381158
```

```
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.3025577771333543
dx error: 7.986982625459595e-09
```

1.6 Implementation of a two-layer NN

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

[30]:
```
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_dims=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 = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
```

```
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
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)
    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: 1.8336562786695002e-08
W2 relative error: 3.201560569143183e-10
b1 relative error: 9.828315204644842e-09
b2 relative error: 4.329134954569865e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.5279152310200606e-07
W2 relative error: 2.8508510893102143e-08
b1 relative error: 1.564679947504764e-08
b2 relative error: 9.089617896905665e-10
```

1.7 Solver

We will now use the cs231n Solver class to train these networks. Familiarize yourself with the API in `cs231n/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 40%.

[37]:
```
model = TwoLayerNet()
solver = None

# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 40%.  We won't have you optimize this further
```

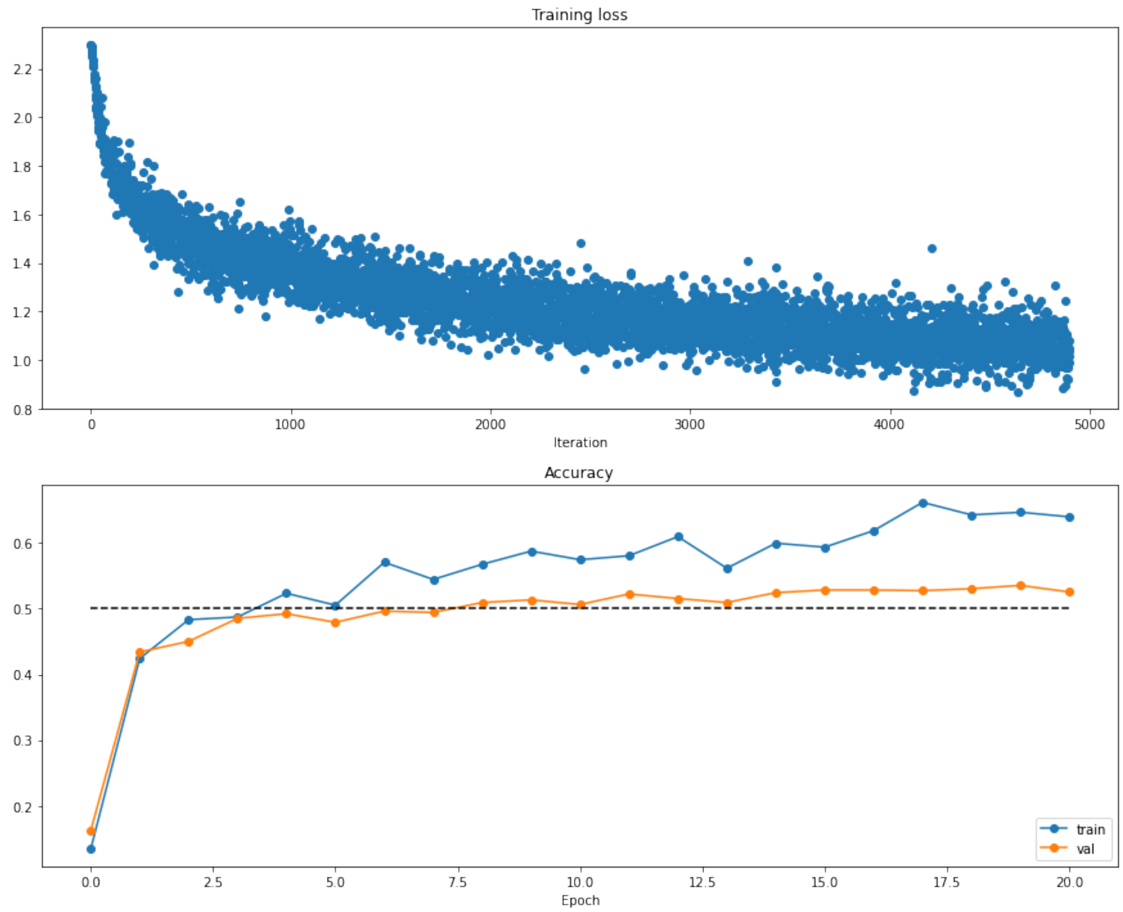
```
#   since you did it in the previous notebook.
#
# ================================================================ #

solver = Solver(model=model, data=data, update_rule='sgd',␣
↪optim_config={'learning_rate' : 1e-3},
                lr_decay=0.90, batch_size=200, num_epochs=20, print_every=100,␣
↪verbose=True,
                num_train_samples=1000, num_val_samples=None,␣
↪checkpoint_name=None)
solver.train()

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

```
(Iteration 1 / 4900) loss: 2.299305
(Epoch 0 / 20) train acc: 0.135000; val_acc: 0.163000
(Iteration 101 / 4900) loss: 1.733263
(Iteration 201 / 4900) loss: 1.623606
(Epoch 1 / 20) train acc: 0.424000; val_acc: 0.434000
(Iteration 301 / 4900) loss: 1.585450
(Iteration 401 / 4900) loss: 1.529780
(Epoch 2 / 20) train acc: 0.483000; val_acc: 0.450000
(Iteration 501 / 4900) loss: 1.449055
(Iteration 601 / 4900) loss: 1.418626
(Iteration 701 / 4900) loss: 1.359526
(Epoch 3 / 20) train acc: 0.487000; val_acc: 0.485000
(Iteration 801 / 4900) loss: 1.331138
(Iteration 901 / 4900) loss: 1.442148
(Epoch 4 / 20) train acc: 0.523000; val_acc: 0.492000
(Iteration 1001 / 4900) loss: 1.441307
(Iteration 1101 / 4900) loss: 1.241867
(Iteration 1201 / 4900) loss: 1.360003
(Epoch 5 / 20) train acc: 0.505000; val_acc: 0.479000
(Iteration 1301 / 4900) loss: 1.215251
(Iteration 1401 / 4900) loss: 1.407873
(Epoch 6 / 20) train acc: 0.570000; val_acc: 0.496000
(Iteration 1501 / 4900) loss: 1.372313
(Iteration 1601 / 4900) loss: 1.240542
(Iteration 1701 / 4900) loss: 1.221818
(Epoch 7 / 20) train acc: 0.544000; val_acc: 0.494000
(Iteration 1801 / 4900) loss: 1.289785
(Iteration 1901 / 4900) loss: 1.203540
(Epoch 8 / 20) train acc: 0.567000; val_acc: 0.509000
(Iteration 2001 / 4900) loss: 1.265486
(Iteration 2101 / 4900) loss: 1.127193
```

```
(Iteration 2201 / 4900) loss: 1.286999
(Epoch 9 / 20) train acc: 0.587000; val_acc: 0.513000
(Iteration 2301 / 4900) loss: 1.155998
(Iteration 2401 / 4900) loss: 1.088092
(Epoch 10 / 20) train acc: 0.574000; val_acc: 0.506000
(Iteration 2501 / 4900) loss: 1.280201
(Iteration 2601 / 4900) loss: 1.295181
(Epoch 11 / 20) train acc: 0.580000; val_acc: 0.522000
(Iteration 2701 / 4900) loss: 1.360415
(Iteration 2801 / 4900) loss: 1.295716
(Iteration 2901 / 4900) loss: 1.237110
(Epoch 12 / 20) train acc: 0.609000; val_acc: 0.515000
(Iteration 3001 / 4900) loss: 1.116251
(Iteration 3101 / 4900) loss: 1.106233
(Epoch 13 / 20) train acc: 0.561000; val_acc: 0.509000
(Iteration 3201 / 4900) loss: 1.150722
(Iteration 3301 / 4900) loss: 1.190257
(Iteration 3401 / 4900) loss: 1.119014
(Epoch 14 / 20) train acc: 0.599000; val_acc: 0.524000
(Iteration 3501 / 4900) loss: 1.086386
(Iteration 3601 / 4900) loss: 1.070148
(Epoch 15 / 20) train acc: 0.593000; val_acc: 0.528000
(Iteration 3701 / 4900) loss: 1.061255
(Iteration 3801 / 4900) loss: 0.975673
(Iteration 3901 / 4900) loss: 1.201995
(Epoch 16 / 20) train acc: 0.618000; val_acc: 0.528000
(Iteration 4001 / 4900) loss: 1.083732
(Iteration 4101 / 4900) loss: 1.017528
(Epoch 17 / 20) train acc: 0.661000; val_acc: 0.527000
(Iteration 4201 / 4900) loss: 1.046494
(Iteration 4301 / 4900) loss: 1.229438
(Iteration 4401 / 4900) loss: 1.060243
(Epoch 18 / 20) train acc: 0.642000; val_acc: 0.530000
(Iteration 4501 / 4900) loss: 1.027325
(Iteration 4601 / 4900) loss: 1.049194
(Epoch 19 / 20) train acc: 0.646000; val_acc: 0.535000
(Iteration 4701 / 4900) loss: 1.071911
(Iteration 4801 / 4900) loss: 1.147977
(Epoch 20 / 20) train acc: 0.639000; val_acc: 0.525000
```

[38]:
```
# 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()
```



1.8 Multilayer Neural Network

Now, we implement a multi-layer neural network.

Read through the `FullyConnectedNet` class in the file `nndl/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 assignment #4.

[58]:
```
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))

  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])))
```

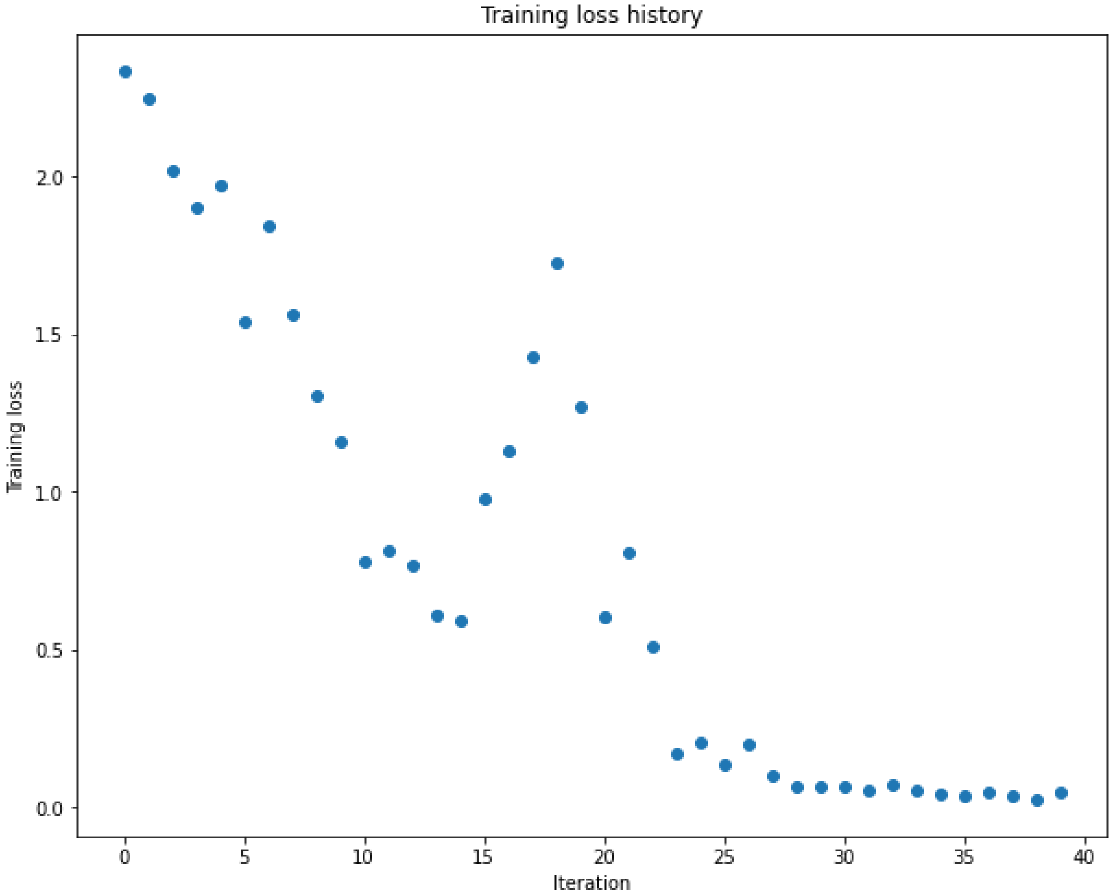
```
Running check with reg = 0
Initial loss: 2.303593768491651
W1 relative error: 7.081455390416797e-07
W2 relative error: 1.7406092190286246e-06
W3 relative error: 4.53909437039025e-08
b1 relative error: 1.1663560611578345e-07
b2 relative error: 2.1920090616659696e-09
b3 relative error: 1.821788214692612e-10
Running check with reg = 3.14
Initial loss: 6.067654822183794
W1 relative error: 1.3167815308043711e-08
W2 relative error: 5.401277175918938e-08
W3 relative error: 1.0
b1 relative error: 2.7027869128619123e-07
b2 relative error: 4.286256654464039e-09
b3 relative error: 1.3276451179013465e-10
```

[61]:
```
# 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.
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 1e-2
learning_rate = 1e-2

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()
```

```
(Iteration 1 / 40) loss: 2.337397
(Epoch 0 / 20) train acc: 0.240000; val_acc: 0.118000
(Epoch 1 / 20) train acc: 0.360000; val_acc: 0.109000
(Epoch 2 / 20) train acc: 0.380000; val_acc: 0.161000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.180000
(Epoch 4 / 20) train acc: 0.700000; val_acc: 0.174000
(Epoch 5 / 20) train acc: 0.680000; val_acc: 0.158000
(Iteration 11 / 40) loss: 0.780584
(Epoch 6 / 20) train acc: 0.860000; val_acc: 0.190000
(Epoch 7 / 20) train acc: 0.640000; val_acc: 0.156000
(Epoch 8 / 20) train acc: 0.720000; val_acc: 0.179000
(Epoch 9 / 20) train acc: 0.640000; val_acc: 0.132000
(Epoch 10 / 20) train acc: 0.780000; val_acc: 0.165000
(Iteration 21 / 40) loss: 0.604248
(Epoch 11 / 20) train acc: 0.900000; val_acc: 0.192000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.186000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.190000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.189000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.194000
(Iteration 31 / 40) loss: 0.064769
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.186000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.195000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.189000
```

```
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.188000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.183000
```



1.9 layers.py

[ ]:

```python
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""
```

```python
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_reshaped = x.reshape(x.shape[0], np.prod(x.shape[1:]))
  out = np.dot(x_reshaped, 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, d1, ..., 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 d1 x ... x dk; 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; it is just the sum over dout examples

  N = dout.shape[0] # 10 , also equal to x.shape[0]
  M = dout.shape[1] # 5
  D = np.prod(x.shape[1:]) # 6

  dx = np.dot(dout, w.T).reshape(x.shape) # (N,M) * (M,D) = (N,D) = (10,6) ->␣
↪reshape to (10, 2, 3)
  dw = np.dot(dout.T, x.reshape(N, D)).T # ((M,N) * (N,D)).T = (D,M) = (6,5)
  db = np.sum(dout, axis=0) # sum down columns/examples of (N,M) matrix -> (M,)␣
↪= (5,)

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

  return dx, dw, db

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

  Input:
  - x: Inputs, of any shape

  Returns a tuple of:
  - out: Output, of the same shape as x
  - cache: x
  """
```

```python
    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ============================================================= #

    out = np.copy(x)
    out[out <= 0] = 0

    # ============================================================= #
    # 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).

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

    Returns:
    - dx: Gradient with respect to x
    """
    x = cache

    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the ReLU backward pass
    # ============================================================= #

    # ReLU directs linearly to those > 0
    x[x<=0] = 0
    x[x>0] = 1
    dx = np.multiply(dout, x)

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

    return dx

def svm_loss(x, y):
    """
```

```
    Computes the loss and gradient using for 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
    """
    N = x.shape[0]
    correct_class_scores = x[np.arange(N), y]
    margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
    margins[np.arange(N), y] = 0
    loss = np.sum(margins) / N
    num_pos = np.sum(margins > 0, axis=1)
    dx = np.zeros_like(x)
    dx[margins > 0] = 1
    dx[np.arange(N), y] -= num_pos
    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)
    N = x.shape[0]
    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
```

1.10 fc_net.py

[ ]:
```
import numpy as np

from .layers import *
from .layer_utils import *

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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 architecure 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_dims=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_dims: 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_dims, num_classes)
    # ================================================================ #

    # randn gives distribution from standardized normal distribution with mean␣
↪0 and variance 1
    self.params['W1'] = np.random.normal(loc=0, scale=weight_scale,␣
↪size=(input_dim, hidden_dims))
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = np.random.normal(loc=0, scale=weight_scale,␣
↪size=(hidden_dims, num_classes))
    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.
    # ================================================================ #

    # cache: (x, w, b)
    out_affine_1_relu, cache_affine_1_relu = affine_relu_forward(X, self.
↪params['W1'], self.params['b1'])
    scores, cache_affine_2 = affine_forward(out_affine_1_relu, self.
↪params['W2'], self.params['b2'])

    # ================================================================ #
    # 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.
    #   i.e., 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, dL = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(np.square(self.params['W1'])) + np.sum(np.
↪square(self.params['W2'])))

    dH, grads['W2'], grads['b2'] = affine_backward(dL, cache_affine_2)
    _, grads['W1'], grads['b1'] = affine_relu_backward(dH, cache_affine_1_relu)
```

```python
        grads['W2'] += self.reg * self.params['W2'] # d(0.5 * reg * (W1**2 +␣
↪W2**2)) / d(W2) = reg * W2
        grads['W1'] += self.reg * self.params['W1'] # d(0.5 * reg * (W1**2 +␣
↪W2**2)) / d(W1) = reg * W1

        # ================================================================ #
        # 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, learnable 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 deteriminstic 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 1 are W1 and b1; and in general the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    # ================================================================ #

    dims = [input_dim] + hidden_dims + [num_classes]
    for i in range(self.num_layers):
        digit = str(i+1)
        self.params['W' + digit] = np.random.normal(0, weight_scale, (dims[i],␣
↪dims[i+1]))
        self.params['b' + digit] = np.zeros(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".
    # ================================================================ #

    # cache: (x, w, b)
    caches = {} # dictionary from layer number to cache objects

    x = X
    for i in range(self.num_layers - 1):
        digit = str(i+1)
        x, caches[digit] = affine_relu_forward(x=x, w=self.params['W' + digit],␣
↪b=self.params['b' + digit])

    # Last layer do affine_forward
    digit = str(self.num_layers)
    scores, caches[digit] = affine_forward(x=x, w=self.params['W' + digit],␣
↪b=self.params['b' + digit])
```

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

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

    loss, grads = 0.0, {}
    # ================================================================ #
    # 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]
    #   Be sure your L2 regularization includes a 0.5 factor.
    # ================================================================ #

    loss, dL = softmax_loss(scores, y)
    reg_loss_sum = 0
    for i in range(self.num_layers - 1):
        reg_loss_sum += np.sum(np.square(self.params['W' + str(i+1)]))
    loss += 0.5 * self.reg * reg_loss_sum

    # First step back do affine_backward: scores, caches[digit]
    digit = str(self.num_layers)
    dx, grads['W' + digit], grads['b' + digit] = affine_backward(dL,␣
↪caches[digit])
    grads['W' + digit] += self.reg * self.params['W' + digit]

    for i in reversed(range(self.num_layers - 1)):
        digit = str(i+1)
        dx, grads['W' + digit], grads['b' + digit] = affine_relu_backward(dx,␣
↪caches[digit])
        grads['W' + digit] += self.reg * self.params['W' + digit]

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return loss, grads
```