1) ECE C147

2) OPTIONAL

two_layer_nn

February 4, 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.

[1]:
```
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))))
```

0.2 Toy example

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

[2]:
```
from nndl.neural_net import TwoLayerNet
```

[3]:
```
# 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
```

```python
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

[4]:
```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

[5]:
```
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)))
```

```
Difference between your loss and correct loss:
0.0
```

[6]:
```
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.

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

```
W2 max relative error: 2.9632227682005116e-10
b2 max relative error: 1.8391748601536041e-10
```

```
W1 max relative error: 1.2832874456864775e-09
b1 max relative error: 3.1726806716844575e-09
```

0.2.4 Training the network

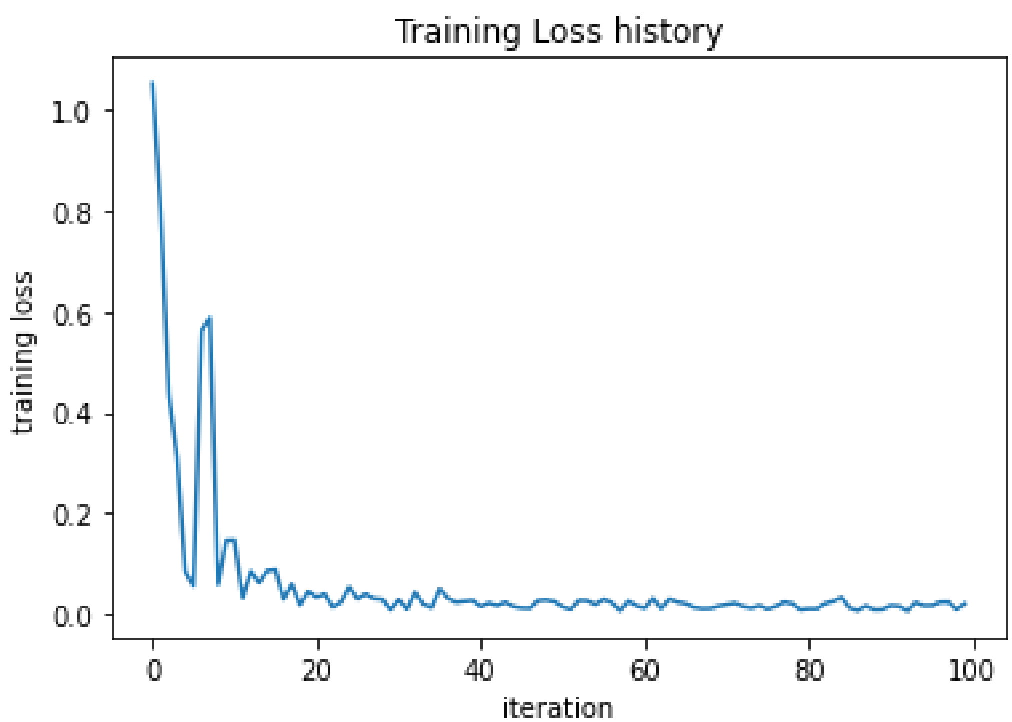
Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax and SVM.

[8]:
```python
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:  0.020371427168192176
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[9]:
```
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 = '../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%.

[10]:
```
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 2.302737768195894
iteration 100 / 1000: loss 2.3022105256991185
iteration 200 / 1000: loss 2.2964000486596787
iteration 300 / 1000: loss 2.268839768313435
iteration 400 / 1000: loss 2.2252579375820387
iteration 500 / 1000: loss 2.153736317283182
iteration 600 / 1000: loss 2.0755138768138344
iteration 700 / 1000: loss 2.021773289161574
iteration 800 / 1000: loss 1.9611502427393566
iteration 900 / 1000: loss 1.9224467427616607
Validation accuracy:  0.282
```

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.

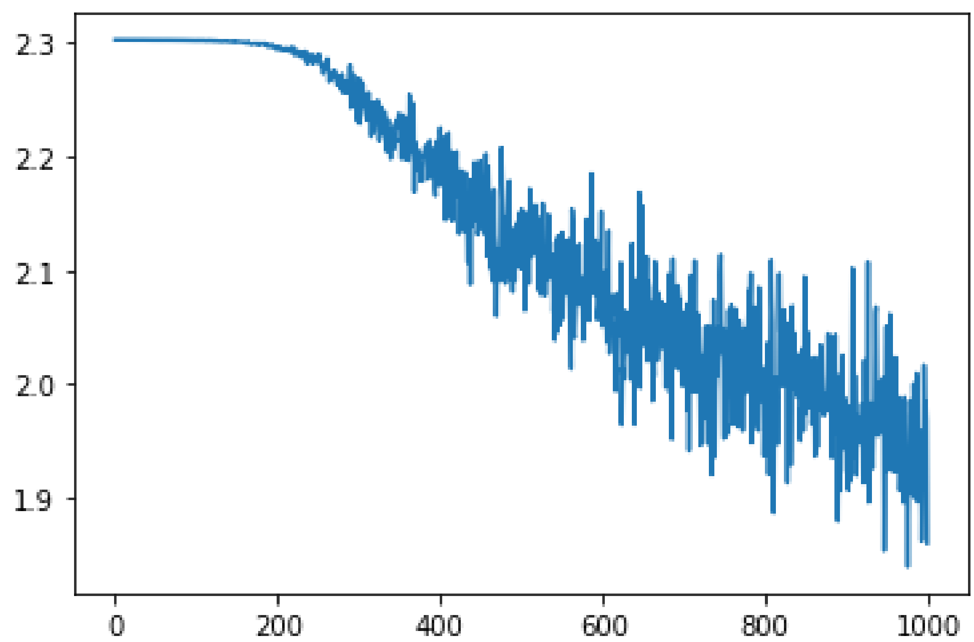
(2) How should you fix the problems you identified in (1)?

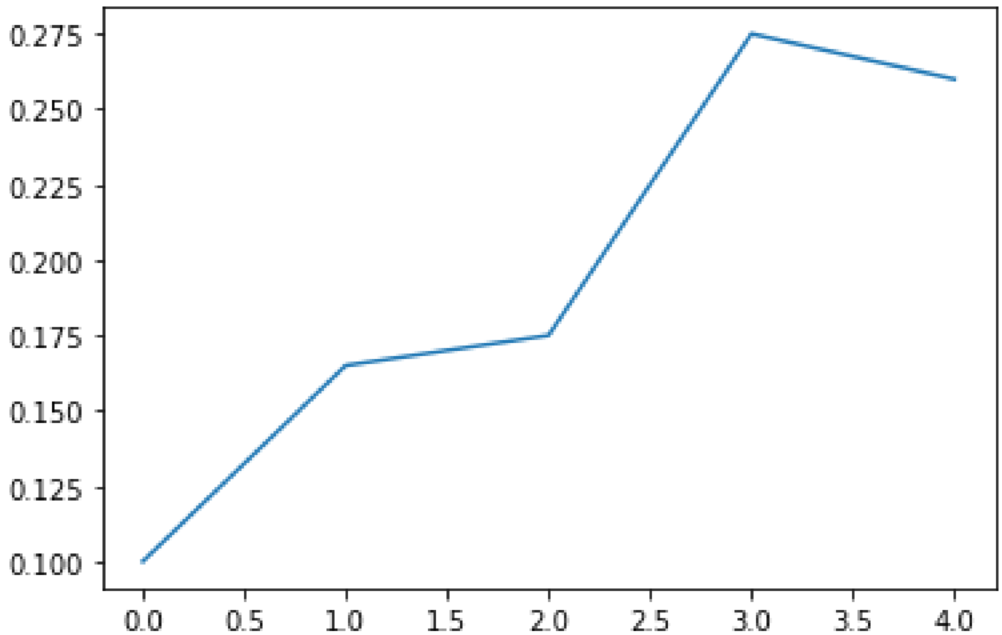
[11]:
```
stats['train_acc_history']
```

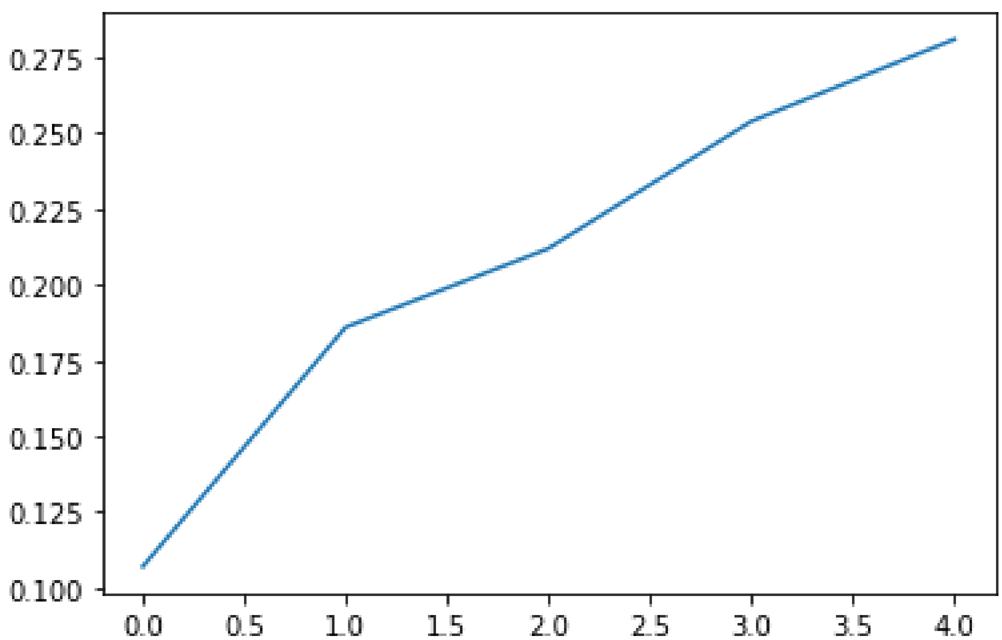
[11]: [0.1, 0.165, 0.175, 0.275, 0.26]

[12]:
```
# ================================================================ #
# 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.show()
plt.plot(stats["train_acc_history"])
plt.show()
plt.plot(stats["val_acc_history"])
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```







0.5 Answers:

(1) Neither loss nor accuracy plateu and are still trending downwards and upwards respectively

(2) This indicates that we are underfitting and so we can increase training iterations, and increase

learning_rate even more to speed up the learning process until it plateus. We can also decrease regularization to help learning occur

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.

[13]:
```
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((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)!
# ================================================================ #
max_val_acc = 0
for num_iters in [5000]:
    for rate in [1e-3, 1e-4, 1e-5]:
        for reg in [0.25, 0.3, 0.35, 0.4]:
            hyperparams = (num_iters, rate, reg)
            net = TwoLayerNet(input_size, hidden_size, num_classes)
            stats = net.train(X_train, y_train, X_val, y_val,
                        num_iters=num_iters, batch_size=200,
                        learning_rate=rate, learning_rate_decay=0.95,
                        reg=reg, verbose=False)
            if stats["val_acc_history"][-1] > max_val_acc:
                max_val_acc = stats["val_acc_history"][-1]
                best_hyper_params = hyperparams
print("Max Val Acc: ", max_val_acc)
print(best_hyper_params)
(num_iters, rate, reg) = best_hyper_params
net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=num_iters, batch_size=200,
            learning_rate=rate, learning_rate_decay=0.95,
            reg=reg, verbose=False)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
best_net = net
```

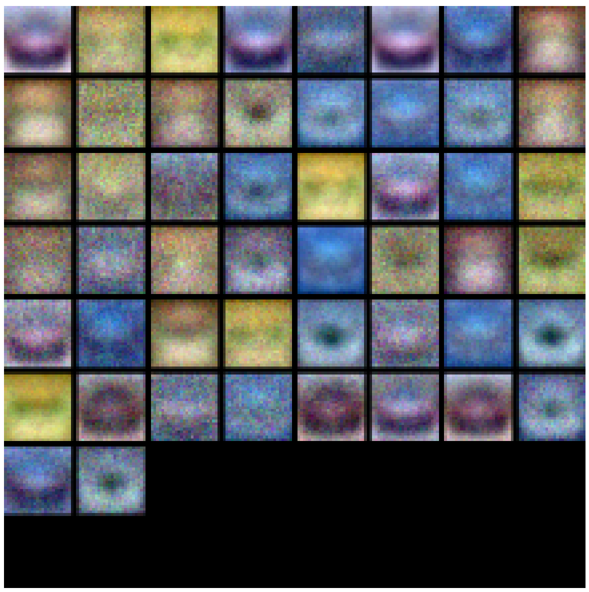
```
Max Val Acc:  0.517
(5000, 0.001, 0.35)
```

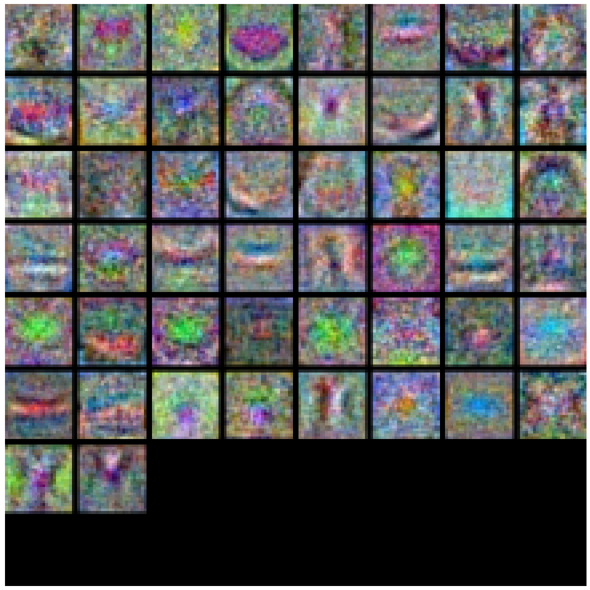
[14]:
```python
from cs231n.vis_utils import visualize_grid

# 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 has much more contrast and defined characteristics for the weights

0.9 Evaluate on test set

[15]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.507
```

[ ]:
```
```

```
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:
        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.
        # ================================================================ #
        h1 = np.maximum(np.dot(X, W1.T) + b1, 0)
        scores = np.dot(h1, W2.T) + b2

        # ================================================================ #
        # 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
        exp_scores = np.exp(scores)
        loss = (
            np.sum(np.log(np.sum(exp_scores, axis=1)) - scores[np.arange(N), y]) / N
        ) + 0.5 * reg * (np.linalg.norm(W1) ** 2 + np.linalg.norm(W2) ** 2)
        # ================================================================ #
        # 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.
        # ================================================================ #
        exp_score_sums = np.sum(exp_scores, axis=1)[:, np.newaxis]
        probs = exp_scores / exp_score_sums
        probs[np.arange(N), y] -= 1

        dL_da = probs / N
        dL_dc2 = dL_da
        dL_db2 = np.sum(dL_da, axis=0)
```

```
        dc2_dW2 = h1
        grads["W2"] = np.dot(dL_dc2.T, dc2_dW2) + (reg * W2)
        grads["b2"] = dL_db2

        dc2_dh1 = W2
        dL_dh1 = (h1 > 0) * np.dot(dL_dc2, dc2_dh1)
        dL_db1 = np.sum(dL_dh1, axis=0)
        dh1_dW1 = X
        dL_dW1 = np.dot(dL_dh1.T, dh1_dW1) + reg * W1
        grads["W1"] = dL_dW1
        grads["b1"] = dL_db1
        # ================================================================ #
        # 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=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.
            # ================================================================ #
            sample_indices = np.random.choice(
                np.arange(num_train), min(num_train, batch_size)
            )
```

```
            X_batch = X[sample_indices, :]
            y_batch = y[sample_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).
            # ================================================================ #
            for param in self.params.keys():
                self.params[param] -= learning_rate * grads[param]

            # ================================================================ #
            # 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.
        # ================================================================ #
        W1, b1 = self.params["W1"], self.params["b1"]
        W2, b2 = self.params["W2"], self.params["b2"]
        h1 = np.maximum(np.dot(W1, X.T) + b1[:, np.newaxis], 0)
```

```
        scores = np.dot(W2, h1) + b2[:, np.newaxis]
        exp_scores = np.exp(scores)
        exp_score_sums = np.sum(exp_scores, axis=0)
        probs = exp_scores / exp_score_sums

        y_pred = np.argmax(probs, axis=0)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return y_pred
```

FC_nets

February 4, 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
```

[1]:

```
## 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))))
```

[2]:
```python
# 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.

[3]:
```python
# 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.

[4]:
```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: 7.598406468756369e-11
dw error: 5.7457383647397536e-11
db error: 1.3332437594154664e-11
```

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.

[5]:
```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.

[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: 3.2756037806455336e-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.

[7]:
```
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: 4.197337774132 45e-11
dw error: 1.0825783318272902e-10
db error: 7.826682563679055e-12
```

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.

[8]:
```
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: 9.00074739665796
```

```
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.3026602945555674
dx error: 8.467349255286659e-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.

[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_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)
```

```python
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: 7.976652806155026e-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%.

[ ]:
```python
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.
#
# ================================================================ #
model = TwoLayerNet(hidden_dims=200)
solver = Solver(model, data, lr_decay=0.9, optim_config={"learning_rate":␣
↪1e-3}, verbose=False)
solver.train()

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

[ ]:
```
# 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.

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

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

[ ]:

[ ]:

```
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.
"""


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

    N = x.shape[0]
    D, M = w.shape
    x_reshaped = np.reshape(x, (N, D))
    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 = x.shape[0]
    D, M = w.shape

    dx = np.reshape(np.dot(dout, w.T), x.shape)
    dw = np.dot(np.reshape(x, (N, D)).T, 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).

    Input:
    - 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.
    # ================================================================== #

    out = np.maximum(0, 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).

    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
    dx = (x > 0) * dout

    # ================================================================ #
    # 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
```

```
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)
```

```
        # ================================================================ #

        self.params["W1"] = np.reshape(
            np.random.normal(0, weight_scale, input_dim * hidden_dims),
            (input_dim, hidden_dims),
        )
        self.params["W2"] = np.reshape(
            np.random.normal(0, weight_scale, hidden_dims * num_classes),
            (hidden_dims, num_classes),
        )
        self.params["b1"] = np.zeros(hidden_dims)
        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.
        # ================================================================ #
        h1, cache1 = affine_relu_forward(X, self.params["W1"], self.params["b1"])
        scores, cache2 = affine_forward(h1, 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, dx = softmax_loss(scores, y)
        loss += (
            0.5
            * self.reg
            * (
                np.linalg.norm(self.params["W1"]) ** 2
                + np.linalg.norm(self.params["W2"]) ** 2
            )
        )
        dh1, grads["W2"], grads["b2"] = affine_backward(dx, cache2)
        _, grads["W1"], grads["b1"] = affine_relu_backward(dh1, cache1)

        grads["W1"] += self.reg * self.params["W1"]
        grads["W2"] += self.reg * self.params["W2"]
        # ================================================================ #
        # 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]
        curr_dim = dims[0]
        next_dim = dims[1]
        for i in range(1, self.num_layers + 1):
            self.params[f"W{i}"] = np.reshape(
                np.random.normal(0, weight_scale, curr_dim * next_dim),
                (curr_dim, next_dim),
            )
            self.params[f"b{i}"] = np.zeros(next_dim)
            curr_dim = next_dim
            next_dim = dims[i]

        # ================================================================ #
        # 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".
        # ================================================================ #
        caches = []
        input = X
        for i in range(1, self.num_layers):
            output, cache = affine_relu_forward(
                input, self.params[f"W{i}"], self.params[f"b{i}"]
            )
            input = output
            caches.append(cache)
        scores, cache = affine_forward(
            input,
            self.params[f"W{self.num_layers}"],
            self.params[f"b{self.num_layers}"],
        )
        caches.append(cache)

        # ================================================================ #
        # 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, dx = softmax_loss(scores, y)
        loss += (
            0.5
            * self.reg
            * sum(
                [
                    np.linalg.norm(self.params[f"W{i}"]) ** 2
                    for i in range(1, self.num_layers + 1)
                ]
            )
        )
        (
            dx,
            grads[f"W{self.num_layers}"],
            grads[f"b{self.num_layers}"],
```

```
        ) = affine_backward(dx, caches[-1])
        for i in range(self.num_layers - 1, 0, -1):
            dx, grads[f"W{i}"], grads[f"b{i}"] = affine_relu_backward(dx, caches[i - 1])

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