1 - FullyConnectedNets

March 20, 2019

1 Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
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
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch Normalization as a tool to more efficiently optimize deep networks.

Acknowledgement: This exercise is adapted from Stanford CS231n.

```
In [1]: # As usual, a bit of setup
        import sys
        import time
        import numpy as np
        import matplotlib.pyplot as plt
        from libs.classifiers.fc_net import *
        from libs.data_utils import get_CIFAR10_data
        from libs.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
        from libs.solver import Solver

        %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))))

        # python setup.py build_ext --inplace
```

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

        data = get_CIFAR10_data()
        for k, v in data.items():
            print('%s: ' % k, v.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,)
```

2 Affine layer: foward

Open the file `libs/layers.py` and implement the `affine_forward` function.
Once you are done you can test your implementaion by running the following:

```
In [3]: # 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: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

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

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

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

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

```
[-0.30972391 -0.66005903 -2.74252641  0.53429886 -0.60239578]
Testing affine_backward function:
dx error:  8.04098721533963e-11
dw error:  8.400644904743146e-10
db error:  3.572414411216823e-11
```

4 ReLU layer: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

```
In [5]: # 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: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU layer: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

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

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

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

```
Testing relu_backward function:
dx error:  3.2756141848873803e-12
```

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `libs/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

```
In [7]: from libs.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, do
        dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, do
        db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, do

        print('Testing affine_relu_forward:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward:
dx error:  2.541199813002474e-10
dw error:  1.5707212103817664e-10
db error:  3.2756318175220628e-12
```

7 Loss layers: Softmax

You implemented this loss function in the last assignment, so we'll give them to you for free here. You should still make sure you understand how it works by looking at the implementations in libs/layers.py.

You can make sure that the implementation is correct by running the following:

```
In [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: 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: ', loss)
        print('dx error: ', rel_error(dx_num, dx))
```

```
Testing softmax_loss:
loss:  2.3025042196092516
dx error:  8.320925333345849e-09
```

8 Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `libs/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

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

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

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

```
print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = 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,
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,
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 = ', 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('%s relative error: %.2e' % (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: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 3.12e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

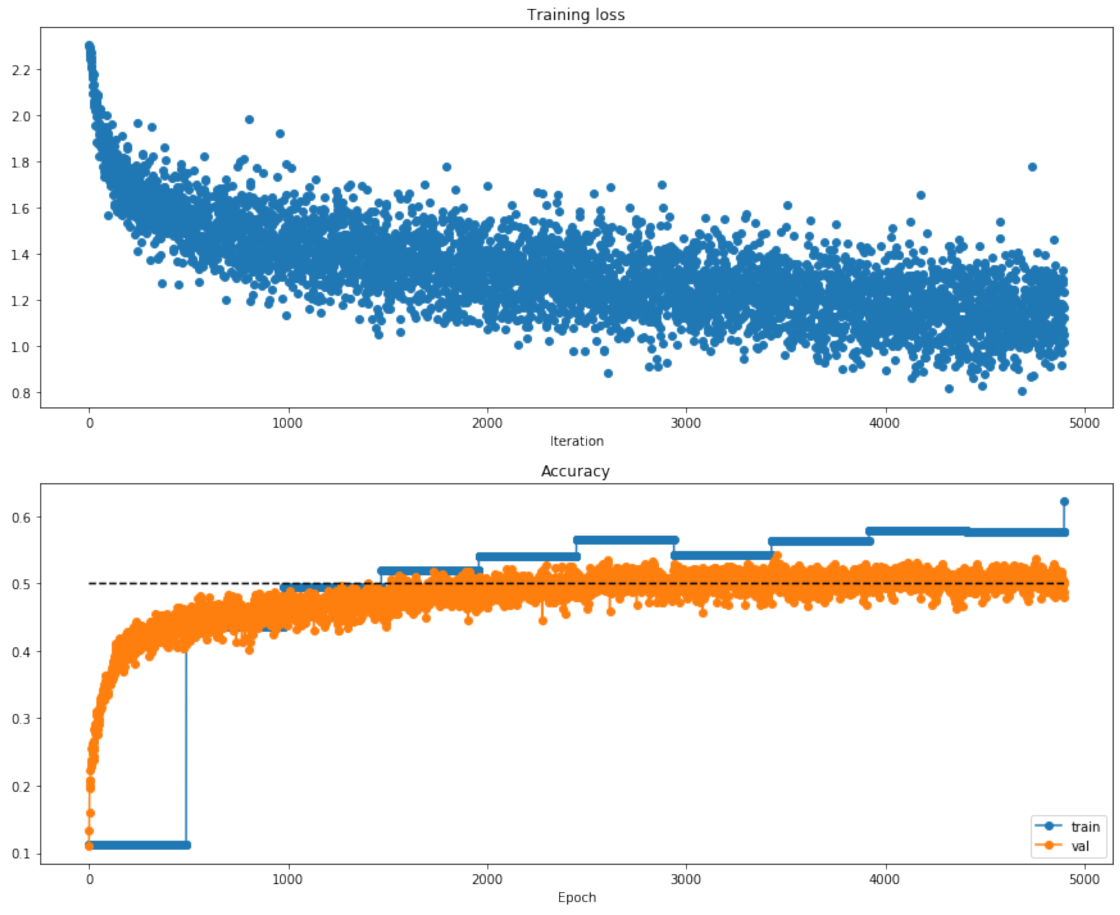
Open the file `libs/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

```
In [10]: # X_val:  (1000, 3, 32, 32)
         # X_train:  (49000, 3, 32, 32)
         # X_test:  (1000, 3, 32, 32)
         # y_val:  (1000,)
         # y_train:  (49000,)
         # y_test:  (1000,)

         # model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

         ##############################################################################
         # TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
         # 50% accuracy on the validation set.                                        #
         ##############################################################################
         model = TwoLayerNet()
         solver = Solver(model, data,
                         update_rule='sgd',
                         optim_config={
                           'learning_rate': 1e-3,
                         },
                         lr_decay=0.95,
                         num_epochs=10, batch_size=100,
                         print_every=1000)
         solver.train()


         ##############################################################################
         #                             END OF YOUR CODE                               #
         ##############################################################################
```

```
(Epoch 0 / 10) (Iteration 1 / 4900) loss: 2.306157 train acc: 0.112000 val_acc: 0.110000
(Epoch 2 / 10) (Iteration 1001 / 4900) loss: 1.441397 train acc: 0.495000 val_acc: 0.467000
(Epoch 4 / 10) (Iteration 2001 / 4900) loss: 1.697318 train acc: 0.541000 val_acc: 0.464000
(Epoch 6 / 10) (Iteration 3001 / 4900) loss: 1.252913 train acc: 0.542000 val_acc: 0.512000
(Epoch 8 / 10) (Iteration 4001 / 4900) loss: 1.020664 train acc: 0.578000 val_acc: 0.511000
```

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

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

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



10 Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `libs/classifiers/fc_net.py`. Implement the initialization, the forward pass, and the backward pass.

10.1 Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-6 or less.

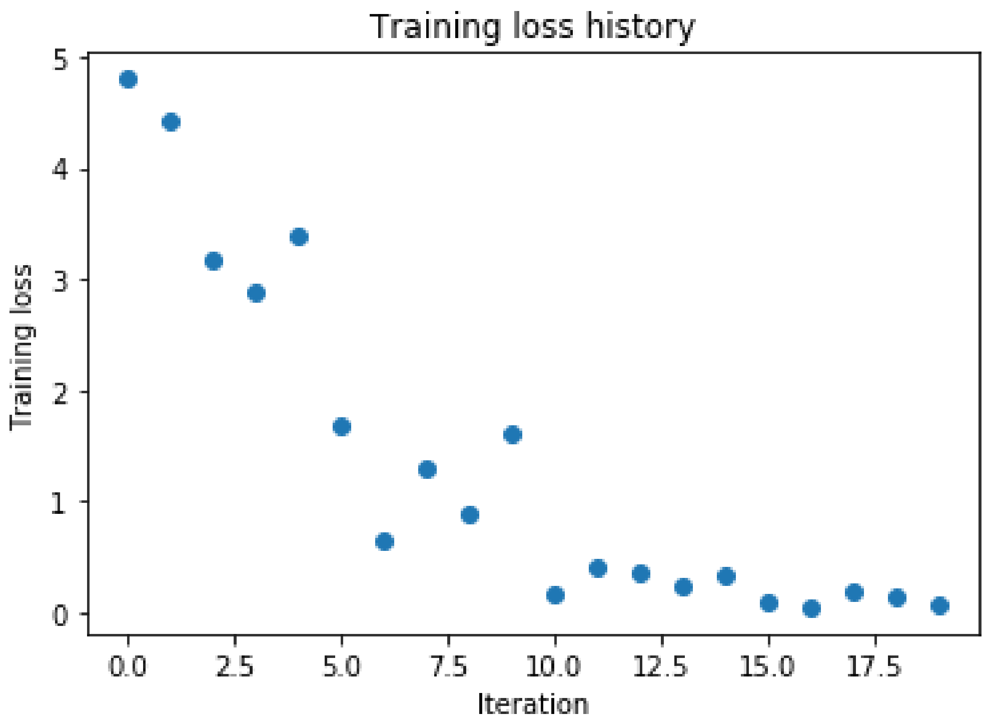
```
In [12]: N, D, H1, H2, C = 2, 15, 20, 30, 10
         X = np.random.randn(N, D)
         y = np.random.randint(C, size=(N,))

         for reg in [0, 3.14]:
             print('Running check with reg = ', 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: ', 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
                 print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.30166474223002
W1 relative error: 2.34e-06
W2 relative error: 1.66e-06
W3 relative error: 6.55e-07
b1 relative error: 1.47e-07
b2 relative error: 5.09e-09
b3 relative error: 8.72e-11
Running check with reg =  3.14
Initial loss:  7.073218960229783
W1 relative error: 1.27e-06
W2 relative error: 1.50e-08
W3 relative error: 5.07e-08
b1 relative error: 6.14e-08
b2 relative error: 1.85e-08
b3 relative error: 1.76e-10
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. You will need to tweak the learning rate and initialization scale, but you should be able to overfit and achieve 100% training accuracy within 20 epochs.

```
In [18]: 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'],
         }

         weight_scale = 0.0278244248968823
         learning_rate = 0.002850013874539408

         ##############################################################################
         # TODO: Use a three-layer Net to overfit 50 training examples.
         ##############################################################################
         model = FullyConnectedNet([100, 100], weight_scale=weight_scale, dtype=np.float64)
         solver = Solver(model, small_data,
                         update_rule='sgd',
                         optim_config={
                           'learning_rate': learning_rate,
                         },
                         lr_decay=0.95,
                         num_epochs=10, batch_size=25,
                         print_every=10)
         solver.train()

         #############################################################################
         #                             END OF YOUR CODE                              #
         ##############################################################################
         plt.plot(solver.loss_history, 'o')
         plt.title('Training loss history')
         plt.xlabel('Iteration')
         plt.ylabel('Training loss')
         plt.show()
```

```
(Epoch 0 / 10) (Iteration 1 / 20) loss: 4.799587 train acc: 0.220000 val_acc: 0.128000
(Epoch 5 / 10) (Iteration 11 / 20) loss: 0.176179 train acc: 0.800000 val_acc: 0.145000
```



In [14]:
```
### tweak the learning rate and initialization scale
learning_rates = 10**np.random.uniform(-3, -2, 5)
weight_scales = 10**np.random.uniform(-2, -1, 5)
best_train = 0
best_pair = None

for lr in learning_rates:
    for ws in weight_scales:
        model = FullyConnectedNet([100, 100], weight_scale=weight_scale, dtype=np.floa
        solver = Solver(model, small_data,
                        update_rule='sgd',
                        optim_config={
                          'learning_rate': learning_rate,
                        },
                        lr_decay=0.95,
                        num_epochs=10, batch_size=25,
                        print_every=10, verbose = False)
        solver.train()
        train_acc = solver.train_acc_history[-1]
        if train_acc > best_train:
            best_train = train_acc
            best_pair = [lr, ws]

print('best training acc:', best_train, 'lr:', best_pair[0], 'ws:', best_pair[1])
```

```
best training acc: 1.0 lr: 0.002850013874539408 ws: 0.0278244248968823
```