FC_nets

February 9, 2024

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).

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:

```python
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
```

[ ]:
```python
## Import and setups

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

%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))))
```

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

[ ]:
```python
# Load the (preprocessed) CIFAR10 data.
# you may find an error here, this is may be because you forgot to use correct␣
 ↪path in get_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.

[ ]:
```
# 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.

```
[ ]: # 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: 4.540760633942363e-10
dw error: 1.2297187173206715e-09
db error: 5.752176884432225e-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.

```
[ ]: # 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.

[ ]:
```
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: 3.2756306009344207e-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.

[ ]:
```
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: 9.066903434772661e-11
dw error: 1.975105316127393e-09
db error: 1.6216672061499507e-11
```

1.5 Softmax loss

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

[ ]:
```
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: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```

Testing softmax_loss:
loss: 2.3022812030082758
dx error: 8.89082935338418e-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.

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

[ ]:
```
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 50%.  We won't have you optimize this further
#   since you did it in the previous notebook.
#
# ================================================================ #

solver = Solver(model, data, update_rule='sgd',
                optim_config={
                    'learning_rate': 1e-3,
                },
                lr_decay=0.95, num_epochs=10,
                batch_size=200, print_every=100
               )


# Train
solver.train()
```

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

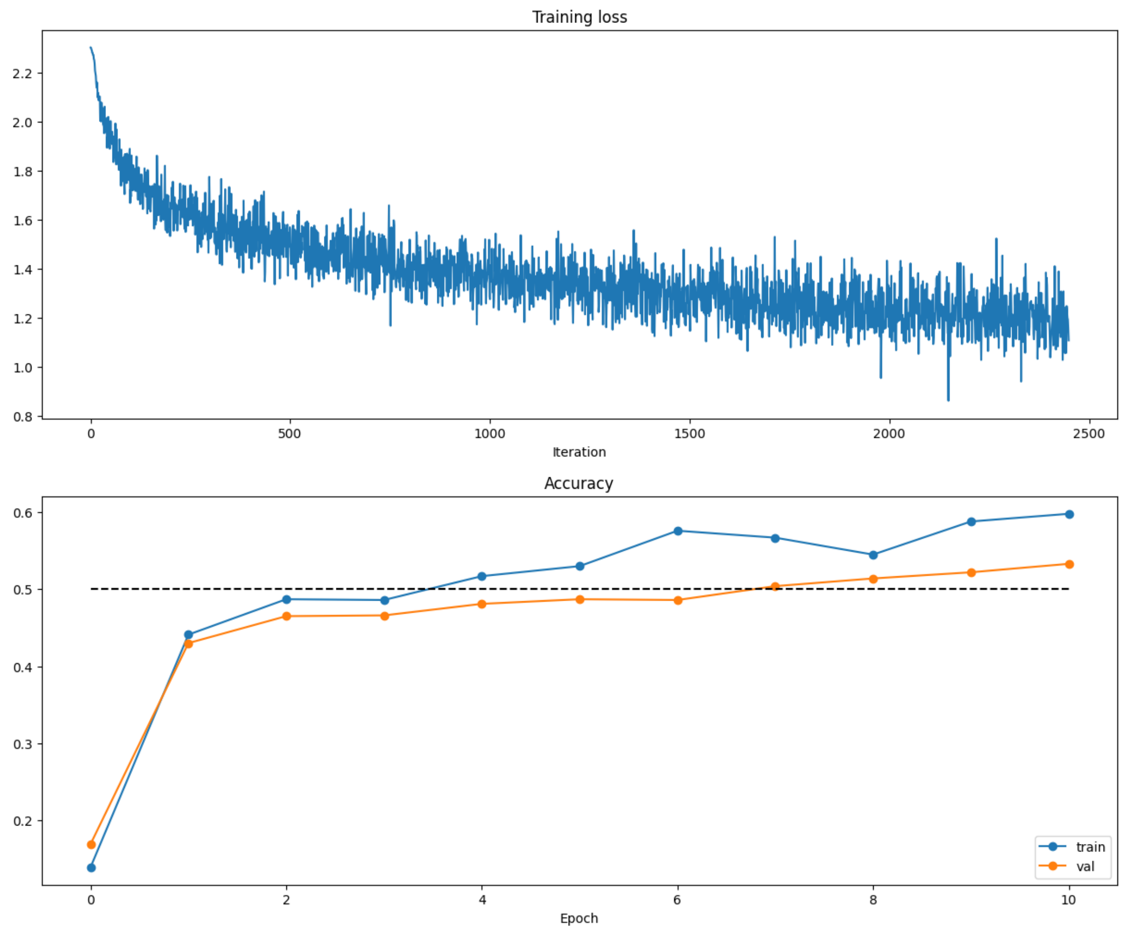
```
(Iteration 1 / 2450) loss: 2.301196
(Epoch 0 / 10) train acc: 0.139000; val_acc: 0.169000
(Iteration 101 / 2450) loss: 1.722966
(Iteration 201 / 2450) loss: 1.607719
(Epoch 1 / 10) train acc: 0.441000; val_acc: 0.430000
(Iteration 301 / 2450) loss: 1.587797
(Iteration 401 / 2450) loss: 1.536156
(Epoch 2 / 10) train acc: 0.487000; val_acc: 0.465000
(Iteration 501 / 2450) loss: 1.559437
(Iteration 601 / 2450) loss: 1.480929
(Iteration 701 / 2450) loss: 1.384908
(Epoch 3 / 10) train acc: 0.486000; val_acc: 0.466000
(Iteration 801 / 2450) loss: 1.350202
(Iteration 901 / 2450) loss: 1.341490
(Epoch 4 / 10) train acc: 0.517000; val_acc: 0.481000
(Iteration 1001 / 2450) loss: 1.276241
(Iteration 1101 / 2450) loss: 1.377728
(Iteration 1201 / 2450) loss: 1.293305
(Epoch 5 / 10) train acc: 0.530000; val_acc: 0.487000
(Iteration 1301 / 2450) loss: 1.354953
(Iteration 1401 / 2450) loss: 1.213575
(Epoch 6 / 10) train acc: 0.576000; val_acc: 0.486000
(Iteration 1501 / 2450) loss: 1.245571
(Iteration 1601 / 2450) loss: 1.217625
(Iteration 1701 / 2450) loss: 1.230809
(Epoch 7 / 10) train acc: 0.567000; val_acc: 0.504000
(Iteration 1801 / 2450) loss: 1.267310
(Iteration 1901 / 2450) loss: 1.211766
(Epoch 8 / 10) train acc: 0.545000; val_acc: 0.514000
(Iteration 2001 / 2450) loss: 1.259551
(Iteration 2101 / 2450) loss: 1.271544
(Iteration 2201 / 2450) loss: 1.193702
(Epoch 9 / 10) train acc: 0.588000; val_acc: 0.522000
(Iteration 2301 / 2450) loss: 1.247702
(Iteration 2401 / 2450) loss: 1.208582
(Epoch 10 / 10) train acc: 0.598000; val_acc: 0.533000
```

[ ]:
```
# 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, '-')
```

```
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 HW #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])))
```

```
Running check with reg = 0
Initial loss: 2.3043988564042834
W1 relative error: 4.026865857833235e-07
W2 relative error: 2.4722178078355698e-06
W3 relative error: 7.786527162866109e-07
b1 relative error: 8.585239097852158e-09
b2 relative error: 6.617393114418291e-09
b3 relative error: 9.094421089313643e-11
Running check with reg = 3.14
Initial loss: 7.088076502489112
W1 relative error: 1.715025677402669e-08
W2 relative error: 1.6416950507237926e-08
W3 relative error: 7.234060596642537e-08
b1 relative error: 3.06595036792245e-08
b2 relative error: 2.3645040815503966e-08
b3 relative error: 1.6171300969446825e-10
```

[ ]:
```
# 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'],
}
```

```python
#### !!!!!!
# 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 = 3e-2
learning_rate = 1e-3

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

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

```
(Iteration 1 / 40) loss: 6.384436
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.120000
(Epoch 1 / 20) train acc: 0.400000; val_acc: 0.128000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.113000
(Epoch 3 / 20) train acc: 0.620000; val_acc: 0.124000
(Epoch 4 / 20) train acc: 0.660000; val_acc: 0.124000
(Epoch 5 / 20) train acc: 0.900000; val_acc: 0.130000
(Iteration 11 / 40) loss: 0.459362
(Epoch 6 / 20) train acc: 0.960000; val_acc: 0.136000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.134000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.126000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.128000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.126000
(Iteration 21 / 40) loss: 0.155344
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.127000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.126000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.129000
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.127000
(Iteration 31 / 40) loss: 0.139198
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.127000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.130000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.128000
```

```
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.130000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.133000
```