Fully connected networks

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

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

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

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

Modular layers

This notebook will build 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
```

In [269…

```
## 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
```

In [270…

```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

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

Linear layers

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

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

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

Affine layer forward pass

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

In [271…

```
# 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
```

Affine layer backward pass

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

In [272…

```
# 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: 1.5039919552889605e-10
dw error: 3.6842057239321546e-11
db error: 3.761595541900145e-11
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

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

In [273…

```
# 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
```

ReLU backward pass

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

In [274…

```
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: 3.2756320364191074e-12
```

Combining the affine and ReLU layers

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

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.

In [275…

```
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: 6.393613414230168e-11
dw error: 4.2773542898028036e-10
db error: 1.5938875168828152e-10
```

Softmax loss

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

In [276…

```
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: 2.3025243225560335
dx error: 9.111196366207372e-09
```

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.

In [277…

```
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
```

Solver

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

In [278…

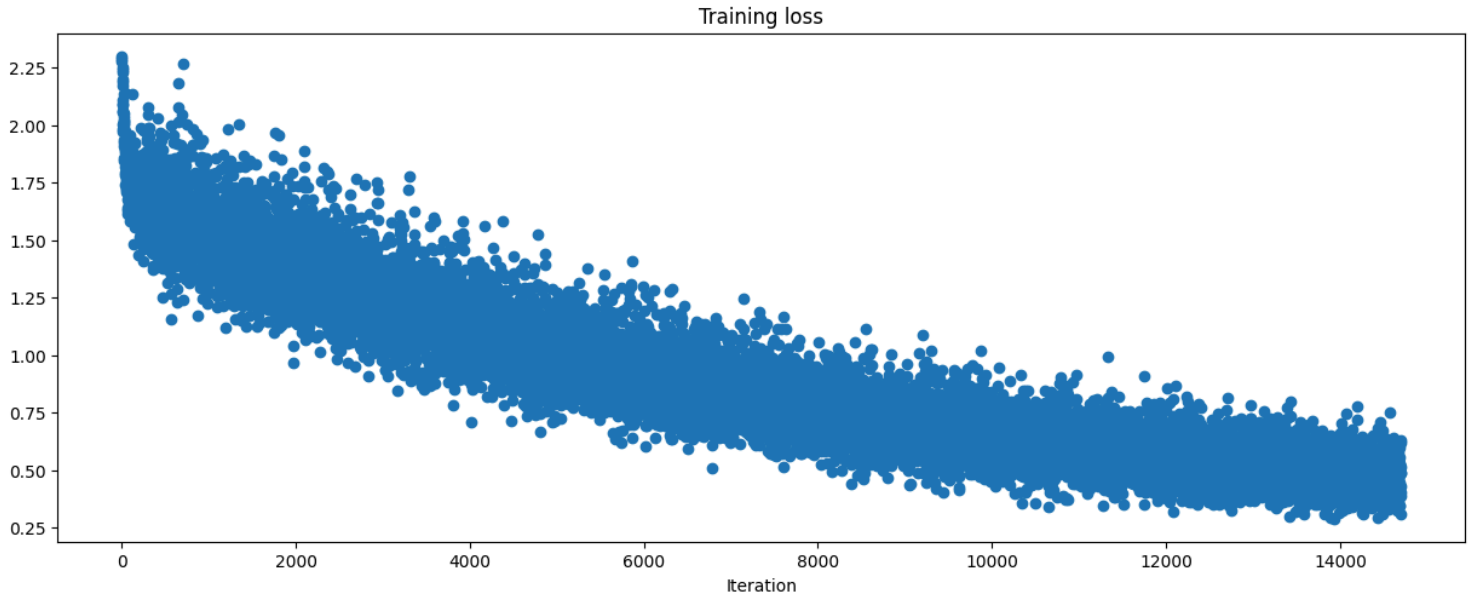
```
model = TwoLayerNet()
solver = None
```

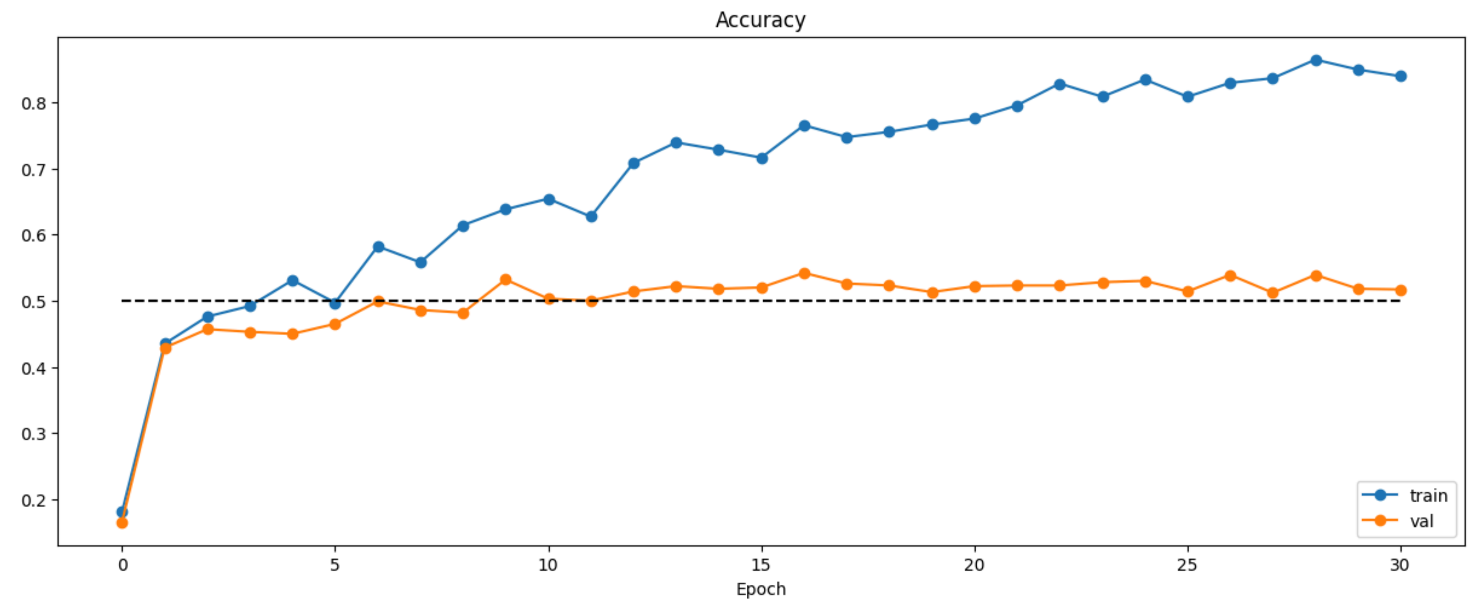
```python
# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 50%.  We won't have you optimize this further
#   since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

model = TwoLayerNet(input_dim = 3*32*32, hidden_dims = 200, num_classes = 10, weight_scale = 1e-3)
solver = Solver(model, data, update_rule = 'sgd', optim_config = {'learning_rate': 0.0018889},
               lr_decay = 0.9125, num_epochs = 30, batch_size = 100, print_every = 100000)
solver.train()

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

```
(Iteration 1 / 14700) loss: 2.299408
(Epoch 0 / 30) train acc: 0.181000; val_acc: 0.166000
(Epoch 1 / 30) train acc: 0.435000; val_acc: 0.429000
(Epoch 2 / 30) train acc: 0.476000; val_acc: 0.457000
(Epoch 3 / 30) train acc: 0.492000; val_acc: 0.453000
(Epoch 4 / 30) train acc: 0.531000; val_acc: 0.450000
(Epoch 5 / 30) train acc: 0.497000; val_acc: 0.465000
(Epoch 6 / 30) train acc: 0.582000; val_acc: 0.499000
(Epoch 7 / 30) train acc: 0.558000; val_acc: 0.486000
(Epoch 8 / 30) train acc: 0.614000; val_acc: 0.482000
(Epoch 9 / 30) train acc: 0.638000; val_acc: 0.532000
(Epoch 10 / 30) train acc: 0.654000; val_acc: 0.503000
(Epoch 11 / 30) train acc: 0.627000; val_acc: 0.500000
(Epoch 12 / 30) train acc: 0.708000; val_acc: 0.514000
(Epoch 13 / 30) train acc: 0.739000; val_acc: 0.522000
(Epoch 14 / 30) train acc: 0.728000; val_acc: 0.518000
(Epoch 15 / 30) train acc: 0.716000; val_acc: 0.520000
(Epoch 16 / 30) train acc: 0.765000; val_acc: 0.542000
(Epoch 17 / 30) train acc: 0.747000; val_acc: 0.526000
(Epoch 18 / 30) train acc: 0.755000; val_acc: 0.523000
(Epoch 19 / 30) train acc: 0.766000; val_acc: 0.513000
(Epoch 20 / 30) train acc: 0.775000; val_acc: 0.522000
(Epoch 21 / 30) train acc: 0.795000; val_acc: 0.523000
(Epoch 22 / 30) train acc: 0.828000; val_acc: 0.523000
(Epoch 23 / 30) train acc: 0.808000; val_acc: 0.528000
(Epoch 24 / 30) train acc: 0.834000; val_acc: 0.530000
(Epoch 25 / 30) train acc: 0.808000; val_acc: 0.514000
(Epoch 26 / 30) train acc: 0.829000; val_acc: 0.539000
(Epoch 27 / 30) train acc: 0.836000; val_acc: 0.512000
(Epoch 28 / 30) train acc: 0.864000; val_acc: 0.539000
(Epoch 29 / 30) train acc: 0.849000; val_acc: 0.518000
(Epoch 30 / 30) train acc: 0.839000; val_acc: 0.517000
```

In [279…

```python
# Run this cell to visualize training loss and train / val accuracy

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

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





Multilayer Neural Network

Now, we implement a multi-layer neural network.

Read through the `FullyConnectedNet` class in the file `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.

In [280…

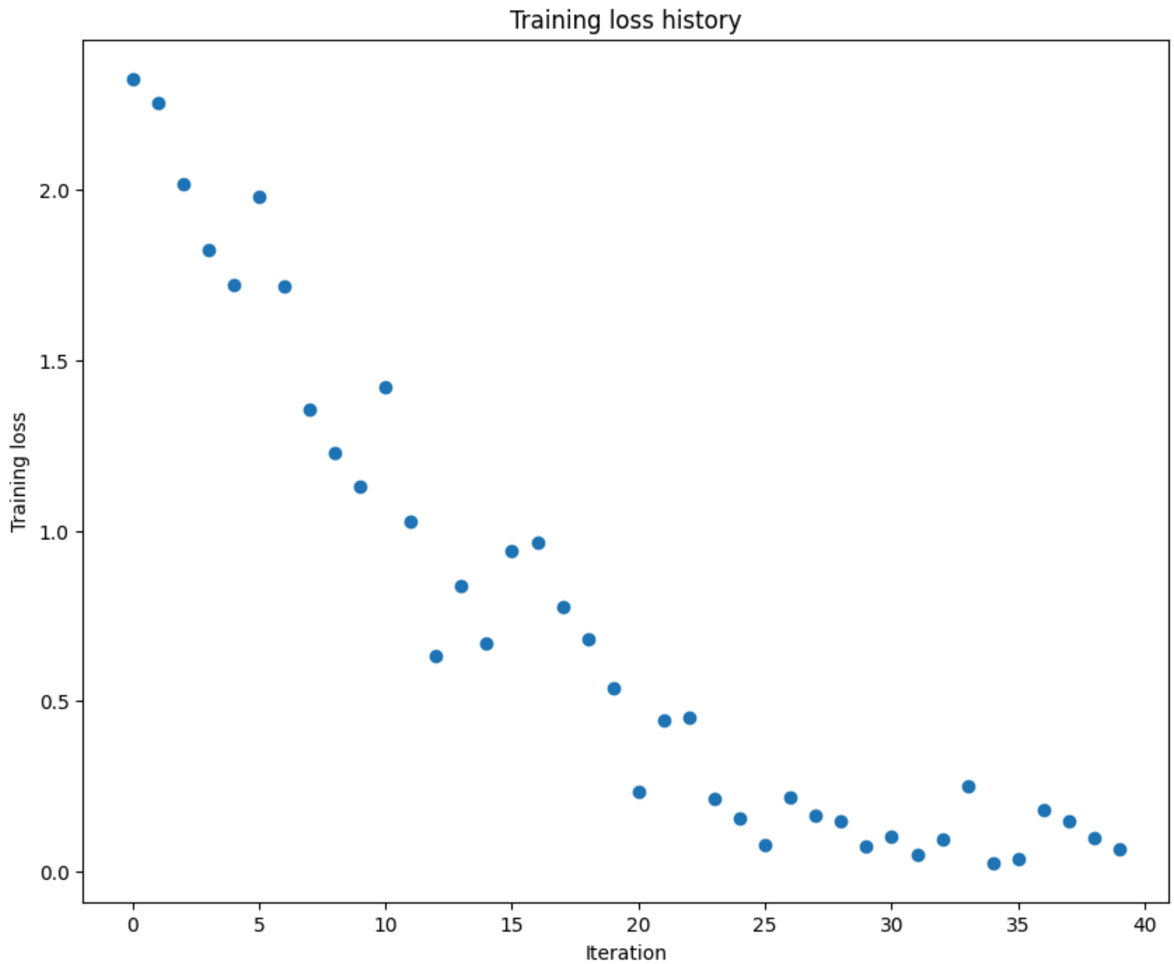
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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: 2.3024388300472083
W1 relative error: 3.1576821093387675e-06
W2 relative error: 1.1091723787052818e-05
W3 relative error: 1.508399576585641e-07
b1 relative error: 1.6091879969487793e-08
b2 relative error: 5.6946467124827834e-08
b3 relative error: 1.0988699277832412e-10
Running check with reg = 3.14
Initial loss: 7.424698263680289
W1 relative error: 7.274884192405169e-08
W2 relative error: 3.174903791964803e-08
W3 relative error: 1.8681267675234432e-08
b1 relative error: 8.522112502637063e-08
b2 relative error: 4.001193954743911e-07
b3 relative error: 2.984230369113938e-10
```

In [281…

```
# Use the three layer neural network to overfit a small dataset.

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


#### !!!!!!
# Play around with the weight_scale and learning_rate so that you can overfit a small dataset.
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 1e-2
learning_rate = 1e-2

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

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

```
(Iteration 1 / 40) loss: 2.325244
(Epoch 0 / 20) train acc: 0.320000; val_acc: 0.148000
(Epoch 1 / 20) train acc: 0.340000; val_acc: 0.107000
(Epoch 2 / 20) train acc: 0.460000; val_acc: 0.176000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.170000
(Epoch 4 / 20) train acc: 0.580000; val_acc: 0.185000
(Epoch 5 / 20) train acc: 0.660000; val_acc: 0.185000
(Iteration 11 / 40) loss: 1.422562
(Epoch 6 / 20) train acc: 0.840000; val_acc: 0.184000
(Epoch 7 / 20) train acc: 0.860000; val_acc: 0.201000
(Epoch 8 / 20) train acc: 0.560000; val_acc: 0.123000
(Epoch 9 / 20) train acc: 0.780000; val_acc: 0.185000
(Epoch 10 / 20) train acc: 0.880000; val_acc: 0.178000
(Iteration 21 / 40) loss: 0.234648
(Epoch 11 / 20) train acc: 0.860000; val_acc: 0.177000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.177000
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.177000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.181000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.186000
(Iteration 31 / 40) loss: 0.105087
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.180000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.181000
(Epoch 18 / 20) train acc: 0.940000; val_acc: 0.183000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.179000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.181000
```



fc_net.py

```
In [ ]: import numpy as np

from .layers import *
from .layer_utils import *


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

  The 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['W2'] = weight_scale * np.random.randn(hidden_dims, num_classes)
    self.params['b2'] = np.zeros(num_classes)

    self.params['W1'] = weight_scale * np.random.randn(input_dim, hidden_dims)
    self.params['b1'] = np.zeros(hidden_dims)

    # ================================================================ #
    # 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, h1_cache = affine_relu_forward(X, self.params['W1'], self.params['b1'])
    scores, scores_cache = 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, ds = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(self.params['W1']**2) + np.sum(self.params['W2']**2))

    dh1, grads['W2'], grads['b2'] = affine_backward(ds, scores_cache)
    grads['W2'] += self.reg * self.params['W2']

    dx, grads['W1'], grads['b1'] = affine_relu_backward(dh1, h1_cache)
    grads['W1'] += self.reg * self.params['W1']

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

    return loss, grads


class FullyConnectedNet(object):
  """
```

```python
    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.
      # ================================================================ #
      layer_dims = np.hstack((input_dim, hidden_dims, num_classes))

      for i in range(1, self.num_layers + 1):
        self.params[f"W{i}"] = weight_scale * np.random.randn(layer_dims[i-1], layer_dims[i])
        self.params[f"b{i}"] = np.zeros(layer_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):
      """
```

```python
    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".
    # ================================================================ #

    H = []
    cache = []
    for i in np.arange(1,self.num_layers + 1):
        Wi, bi = f'W{i}', f'b{i}'

        if i == 1:
            H.append(affine_relu_forward(X, self.params[Wi], self.params[bi])[0])
            cache.append(affine_relu_forward(X, self.params[Wi], self.params[bi])[1])

        elif i == self.num_layers:
            scores = affine_forward(H[i-2], self.params[Wi], self.params[bi])[0]
            cache.append(affine_forward(H[i-2], self.params[Wi], self.params[bi])[1])

        else:
            H.append(affine_relu_forward(H[i-2], self.params[Wi], self.params[bi])[0])
            cache.append(affine_relu_forward(H[i-2], self.params[Wi], self.params[bi])[1])



    # ================================================================ #
    # 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, dz = softmax_loss(scores, y)
    dh = dz
    grads = {}

    for i in range(self.num_layers, 0, -1):
        Wi, bi = f'W{i}', f'b{i}'
        loss += 0.5 * self.reg * np.sum(self.params[Wi] ** 2)

        # Backpropagation
        if i == self.num_layers:
            dh, grads[Wi], grads[bi] = affine_backward(dh, cache[-1])
        else:
            dh, grads[Wi], grads[bi] = affine_relu_backward(dh, cache[i-1])

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



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

layers.py

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

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

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

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

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """

  # ================================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ================================================================ #

  x_reshaped = x.reshape(x.shape[0], -1)
  out = 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

  x_reshaped = x.reshape(x.shape[0], -1)
  db = np.sum(dout, axis = 0)
  dw = x_reshaped.T @ dout
  dx = np.dot(dout, w.T).reshape(x.shape)

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

  return dx, dw, db

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

  Input:
  - x: Inputs, of any shape

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

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

    relu = lambda x: x * (x > 0)
    out = relu(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
    # ================================================================ #

    dx = dout * (x > 0)

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

    return 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
```