FC Net

In this section all relevant python code files will be displayed in their entirety.
Each file will be labeled accordingly.

fc_net.py

```
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
```

```python
    # ================================================================ #
    # 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.random.randn(input_dim,hidden_dims)*weight_scale
    self.params['b1'] = np.zeros((hidden_dims,))
    self.params['W2'] = np.random.randn(hidden_dims,num_classes)*weight_scale
    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.
    # ================================================================ #
```

```python
    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,dLdz = softmax_loss(scores,y)
    loss += 0.5*self.reg*np.linalg.norm(self.params['W1'])**2 +0.5*self.reg*np.linalg.norm(s
    dLdh1,dLdw2, dLdb2 = affine_backward(dLdz,cache2)
    dLdx,dLdw1,dLdb1 = affine_relu_backward(dLdh1,cache1)
    grads['W2'] = dLdw2 + self.reg*self.params['W2']
    grads['b2'] = dLdb2
    grads['W1'] = dLdw1 + self.reg*self.params['W1']
    grads['b1'] = dLdb1
    # ================================================================ #
    # 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
```

```python
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    # ================================================================ #

    for i in range(self.num_layers):
      w_i = 'W'+str(i+1)
      b_i = 'b'+str(i+1)
      if(i==0):
        self.params[w_i] = np.random.randn(input_dim,hidden_dims[i])*weight_scale
        self.params[b_i] = np.zeros((hidden_dims[i],))
      elif(i==self.num_layers-1):
        self.params[w_i] = np.random.randn(hidden_dims[i-1],num_classes)*weight_scale
        self.params[b_i] = np.zeros((num_classes,))
      else:
        self.params[w_i] = np.random.randn(hidden_dims[i-1],hidden_dims[i])*weight_scale
        self.params[b_i] = np.zeros((hidden_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".
# ================================================================ #
hs = []
caches= []
for i in range(self.num_layers):
  #generate keys
  w_i = 'W'+str(i+1)
  b_i = 'b'+str(i+1)
  if(i==0):
    h_i, cache = affine_relu_forward(X,self.params[w_i], self.params[b_i])
    hs.append(h_i)
    caches.append(cache)
  elif(i!= self.num_layers-1):
    h_i, cache = affine_relu_forward(hs[i-1],self.params[w_i], self.params[b_i])
    hs.append(h_i)
    caches.append(cache)
  else:
    h_i,cache = affine_forward(hs[i-1],self.params[w_i], self.params[b_i])
    scores = h_i
    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,dLdupstream = softmax_loss(scores,y)
    dLdhi,dLdwi, dLdbi =0,0,0
    for i in reversed(range(self.num_layers)):
      #generate keys
      w_i = 'W'+str(i+1)
      b_i = 'b'+str(i+1)
      loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2
      if(i == self.num_layers-1):
        dLdhi,dLdwi, dLdbi = affine_backward(dLdupstream,caches[i])
      else:
        dLdhi,dLdwi,dLdbi = affine_relu_backward(dLdupstream,caches[i])
      dLdupstream = dLdhi
      grads[w_i]=dLdwi + self.reg*self.params[w_i]
      grads[b_i]=dLdbi

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

layers.py

```
import numpy as np
import pdb




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.
```

```python
    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_shapes = x.shape
    D = np.prod(x_shapes[1:])
    N = x_shapes[0]
    X = x.reshape(N,D)
    out = X@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,)
    """
```

```python
    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
    # dw should be D x M; it relates to dout through multiplication with x, which is N x D aft
    # db should be M; it is just the sum over dout examples
    x_shapes = x.shape
    D = np.prod(x_shapes[1:])
    N = x_shapes[0]
    x_reshape = x.reshape(N,D)
    dx = (dout@w.T).reshape(x_shapes)
    dw = x_reshape.T@dout
    db = dout.T@np.ones((N,))
    # ================================================================ #
    # 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 = (x>0)*x
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
```

```python
    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 = dout*(x>0)

    # ================================================================ #
    # 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]
```

```python
  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
```