fc_net.py

This file contains all the code for fc_net.py

Code

```
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)
        # ================================================================ #


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

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

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

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

    return loss, grads


class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden layers,
  ReLU nonlinearities, and a softmax loss function. This will also implement
  dropout and batch normalization as options. For a network with L layers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

  where batch normalization and dropout are optional, and the {...} block is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
```

```
"""
Initialize a new FullyConnectedNet.

Inputs:
- hidden_dims: A list of integers giving the size of each hidden layer.
- input_dim: An integer giving the size of the input.
- num_classes: An integer giving the number of classes to classify.
- dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
  the network should not use dropout at all.
- use_batchnorm: Whether or not the network should use batch normalization.
- reg: Scalar giving L2 regularization strength.
- weight_scale: Scalar giving the standard deviation for random
  initialization of the weights.
- dtype: A numpy datatype object; all computations will be performed using
  this datatype. float32 is faster but less accurate, so you should use
  float64 for numeric gradient checking.
- seed: If not None, then pass this random seed to the dropout layers. This
  will make the dropout layers deteriminstic so we can gradient check the
  model.
"""
self.use_batchnorm = use_batchnorm
self.use_dropout = dropout > 0
self.reg = reg
self.num_layers = 1 + len(hidden_dims)
self.dtype = dtype
self.params = {}

# ================================================================ #
# YOUR CODE HERE:
#   Initialize all parameters of the network in the self.params dictionary.
#   The weights and biases of layer 1 are W1 and b1; and in general the
#   weights and biases of layer i are Wi and bi. The
#   biases are initialized to zero and the weights are initialized
#   so that each parameter has mean 0 and standard deviation weight_scale.
#
#   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
#   parameters to zero.  The gamma and beta parameters for layer 1 should
#   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
#   should be gamma2 and beta2, etc. Only use batchnorm if self.use_batchnorm
#   is true and DO NOT do batch normalize the output scores.
# ================================================================ #
for i in range(self.num_layers):
  w_i = 'W'+str(i+1)
  b_i = 'b'+str(i+1)
  gamma_i = 'gamma'+str(i+1)
  beta_i = 'beta'+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],))
        if(self.use_batchnorm):
          self.params[gamma_i] = np.ones((hidden_dims[i],))
          self.params[beta_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],))
        if(self.use_batchnorm):
          self.params[gamma_i] = np.ones((hidden_dims[i],))
          self.params[beta_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".
#
#   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
#   between the affine_forward and relu_forward layers.  You may
#   also write an affine_batchnorm_relu() function in layer_utils.py.
#
#   DROPOUT: If dropout is non-zero, insert a dropout layer after
#   every ReLU layer.
# ================================================================ #
hs = []
caches= []
dro_caches = []
for i in range(self.num_layers):
  #generate keys
  w_i = 'W'+str(i+1)
  b_i = 'b'+str(i+1)
  gamma_i = 'gamma'+str(i+1)
  beta_i = 'beta'+str(i+1)
  if(i==0):
    if(self.use_batchnorm):
      h_i, cache = affine_batchnorm_relu_forward(X,self.params[w_i], self.params[b_i], s
      if self.use_dropout:
        h_i, dro_cache = dropout_forward(h_i,self.dropout_param)
        dro_caches.append(dro_cache)
      hs.append(h_i)
      caches.append(cache)
```

```
      else:
        h_i, cache = affine_relu_forward(X,self.params[w_i], self.params[b_i])
        if self.use_dropout:
          h_i, dro_cache = dropout_forward(h_i,self.dropout_param)
          dro_caches.append(dro_cache)
        hs.append(h_i)
        caches.append(cache)

    elif(i!= self.num_layers-1):
      if(self.use_batchnorm):
        h_i, cache = affine_batchnorm_relu_forward(hs[i-1],self.params[w_i], self.params[b
        if self.use_dropout:
          h_i, dro_cache = dropout_forward(h_i,self.dropout_param)
          dro_caches.append(dro_cache)
        hs.append(h_i)
        caches.append(cache)

      else:
        h_i, cache = affine_relu_forward(hs[i-1],self.params[w_i], self.params[b_i])
        if self.use_dropout:
          h_i, dro_cache = dropout_forward(h_i,self.dropout_param)
          dro_caches.append(dro_cache)
        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.
  #
  #   BATCHNORM: Incorporate the backward pass of the batchnorm.
  #
```

```
#   DROPOUT: Incorporate the backward pass of dropout.
# ================================================================ #
loss,dLdupstream = softmax_loss(scores,y)
dLdhi,dLdwi, dLdbi, dLdgammai, dLdbetai =0,0,0,0,0
for i in reversed(range(self.num_layers)):
  #generate keys
  w_i = 'W'+str(i+1)
  b_i = 'b'+str(i+1)
  gamma_i = 'gamma'+str(i+1)
  beta_i = 'beta'+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:
    if(self.use_batchnorm):
      if self.use_dropout:
        dLdupstream = dropout_backward(dLdupstream,dro_caches[i])
      dLdhi,dLdwi,dLdbi, dLdgammai, dLdbetai = affine_batchnorm_relu_backward(dLdupstrea
      grads[gamma_i] = dLdgammai
      grads[beta_i] = dLdbetai
    else:
      if self.use_dropout:
        dLdupstream = dropout_backward(dLdupstream,dro_caches[i])
      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
```