fc_net

February 10, 2023

[ ]:
```
# %load fc_net.py
import numpy as np
import pdb

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=1, 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)
        # ================================================================ #
        W1, W2 =  self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        output_first_layer, cache_first = affine_relu_forward(X, W1, b1)
        scores, cache_sec = affine_forward(output_first_layer, W2,b2)

        # ================================================================ #
        # 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
        # ================================================================ #

        # 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=1, 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=1␣
↪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 < 1
        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.
        #
        #   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.
        # ================================================================
        hidd_len = len(hidden_dims)
        shape_one = input_dim
        for i in range(hidd_len):
            self.params['W'+str(i+1)] = np.random.randn(shape_one,␣
↪hidden_dims[i]) * weight_scale
            self.params['b'+str(i+1)] = np.zeros(hidden_dims[i])
            shape_one = hidden_dims[i]
            if self.use_batchnorm is True:
                self.params['gamma'+str(i+1)] = np.ones(hidden_dims[i])
                self.params['beta'+str(i+1)] = np.zeros(hidden_dims[i])
        self.params['W'+str(hidd_len+1)] = np.random.randn(shape_one,␣
↪num_classes) * weight_scale
        self.params['b'+str(hidd_len+1)] = np.zeros(num_classes)
        pass

        # ================================================================ #
        # 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.
        # ================================================================ #
        output = {}
```

```python
    output[0] = X
    cache = {}
    for i in range(self.num_layers):
      j = i + 1
      weights = str('W') + str(j)
      bias = str('b') + str(j)

      W_j = self.params[weights]
      b_j = self.params[bias]

      if j == self.num_layers:
        output[j], cache[i] = affine_forward(output[i], W_j, b_j)
      else:
        if self.use_dropout and self.use_batchnorm:
          gammas = str('gamma') + str(j)
          betas = str('beta') + str(j)
          gamma_j = self.params[gammas]
          beta_j = self.params[betas]
          output[j], cache[i] = affine_batchnorm_relu_dropout_forward(output[i], W_j, b_j, gamma_j, beta_j, self.bn_params[i], self.dropout_param)

        elif self.use_batchnorm is True:
          gammas = str('gamma') + str(j)
          betas = str('beta') + str(j)
          gamma_j = self.params[gammas]
          beta_j = self.params[betas]
          output[j], cache[i] = affine_batchnorm_relu_forward(output[i], W_j, b_j, gamma_j, beta_j, self.bn_params[i])

        else:
          output[j], cache[i] = affine_relu_forward(output[i], W_j, b_j)

    scores = output[self.num_layers]
    pass

    # ================================================================ #
    # 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, dout = softmax_loss(scores, y)
        reg_loss = 0
        for i in range(self.num_layers):
          j = i + 1
          weights = str('W') + str(j)
          W_j = self.params[weights]
          reg_loss = reg_loss + 0.5 * self.reg * np.sum(W_j * W_j)
        loss = loss + reg_loss

        #compute grad
        hidd_len = self.num_layers-1
        # weights and bias of the last layer
        dx, last_weight, last_bias = affine_backward(dout, cache[self.
↪num_layers-1])
        grads['W'+str(self.num_layers)] = last_weight + self.reg * self.
↪params['W'+str(self.num_layers)]
        grads['b'+str(self.num_layers)] = last_bias
        # weights, bias, gamma, beta of the rest layers
        if self.use_dropout and self.use_batchnorm:
          for i in range(hidd_len, 0, -1):
                dx, dw, db, dgamma, dbeta =␣
↪affine_batchnorm_relu_dropout_backward(dx, cache[i-1])
                grads['W'+str(i)] = dw + self.reg * self.params['W'+str(i)]
                grads['b'+str(i)] = db
                grads['gamma'+str(i)] = dgamma
                grads['beta'+str(i)] = dbeta
        elif self.use_batchnorm is True:
          for i in range(hidd_len, 0, -1):
                dx, dw, db, dgamma, dbeta = affine_batchnorm_relu_backward(dx,␣
↪cache[i-1])
                grads['W'+str(i)] = dw + self.reg * self.params['W'+str(i)]
                grads['b'+str(i)] = db
                grads['gamma'+str(i)] = dgamma
                grads['beta'+str(i)] = dbeta
        else:
          for i in range(self.num_layers-1, 0, -1):
                dx, dw, db = affine_relu_backward(dx, cache[i-1])
```

```
                grads['W'+str(i)] = dw + self.reg * self.params['W'+str(i)]
                grads['b'+str(i)] = db

        pass

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

        return loss, grads
```