

```

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 architecture 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['W1'] = np.random.normal(loc=0.0, scale=weight_scale, size=(input_dim,
hidden_dims))
        self.params['b1'] = 0
        self.params['W2'] = np.random.normal(loc=0.0, scale=weight_scale, size=(hidden_dims,
num_classes))
        self.params['b2'] = 0

        # print(self.params['W1'].shape)
        # print(self.params['W2'].shape)

        pass

        # ===== #
        # 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, \dots, 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.

=====

W1, b1 = self.params['W1'], self.params['b1']

W2, b2 = self.params['W2'], self.params['b2']

out, cache = affine_relu_forward(x, w, b)

dx, dw, db = affine_relu_backward(dout, cache)

h1_relu, h1_cache = affine_relu_forward(X, W1, b1) #cache = (fc_cache, relu_cache)

h2, h2_forward_cache = affine_forward(h1_relu, W2, b2)

scores = h2

pass

=====

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 $0.5 * \text{self.reg} * W^2$
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, dscores = softmax_loss(scores, y)

l2_reg = 0.5 * self.reg * (np.sum(W1 * W1) + np.sum(W2 * W2))

loss += l2_reg

print(loss)

```

dx, grads['W2'], grads['b2'] = affine_backward(dscores, h2_forward_cache)
dx, grads['W1'], grads['b1'] = affine_relu_backward(dx, h1_cache)

grads['W1'] += self.reg * W1
grads['W2'] += self.reg * W2
pass

# ===== #
# 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 deterministic 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 = {}
self.hidden_dim = hidden_dims

```

```

# ===== #

```

```

# 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.
# ===== #

# print(self.num_layers)

# Initialize parameters for each layer
layer_input_dim = input_dim
for i, hd in enumerate(hidden_dims):
    # Weight matrix
    self.params[f'W{i + 1}'] = weight_scale * np.random.randn(layer_input_dim, hd)
    # Bias vector
    self.params[f'b{i + 1}'] = np.zeros(hd)

    layer_input_dim = hd # Update input dimension for the next layer

# Initialize parameters for the output layer
self.params[f'W{self.num_layers}'] = weight_scale * np.random.randn(layer_input_dim,
num_classes)
self.params[f'b{self.num_layers}'] = np.zeros(num_classes)

# print(self.params.keys())

# ===== #
# 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".
# ===== #
affine_relu_caches = []
X_temp = X
for i in range(1, self.num_layers):
    W, b = self.params[f'W{i}'], self.params[f'b{i}']
    # print(i)
    X_temp, cache = affine_relu_forward(X_temp, W, b)
    affine_relu_caches.append(cache)

    scores, output_cache = affine_forward(X_temp, self.params[f'W{self.num_layers}'],
self.params[f'b{self.num_layers}'])

# ===== #
# 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, dscores = softmax_loss(scores, y)
l2_reg = 0
# add in regularized contributions for each weight matrix
for i in range(1, self.num_layers+1):
    W = self.params[f'W{i}']
    l2_reg += 0.5 * self.reg * np.sum(W * W)
loss += l2_reg

#last layer gradient
dx, grads[f'W{self.num_layers}'], grads[f'b{self.num_layers}'] = affine_backward(dscores,
output_cache)

# for i in range(1, self.num_layers):
#     print(i)
#     dx, grads[f'W{i}'], grads[f'b{i}'] = affine_relu_backward(dx, affine_relu_caches[i-
1])

for i in range(self.num_layers - 1, 0, -1):
    dx, grads[f'W{i}'], grads[f'b{i}'] = affine_relu_backward(dx, affine_relu_caches[i -
1])

for i in range(1, self.num_layers+1):
    grads[f'W{i}'] += self.reg * self.params[f'W{i}']

pass

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

```