In [ ]:

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

        #we will Initialize them based on layers:
        for i in np.arange(1, self.num_layers + 1): #for all layers

            if i == 1: #for the layer 1

                self.params['b1'] = np.zeros(hidden_dims[i - 1]) #based on the size of the first hidden layer
                #the weights will have mean 0 and standard deviation weight_scale
                self.params['W1'] = np.random.normal(0, weight_scale, (input_dim, hidden_dims[i - 1]))

                #if the network uses batch normalization
                if self.use_batchnorm == True:
                    self.params['gamma1'] = np.ones(hidden_dims[i - 1]) #the gammas should be 1
                    self.params['beta1'] = np.zeros(hidden_dims[i - 1]) #the betas should be zero

            elif i == self.num_layers: #for the last layer

                #the size after the last layer will be the number of classes
                self.params['b' + str(i)] = np.zeros(num_classes)
                self.params['W' + str(i)] = np.random.normal(0, weight_scale, (hidden_dims[i - 2], num_classes))

            else: #for the layer except the first and the last

                #based on the size of the ith hidden layer
```

```
                self.params['b' + str(i)] = np.zeros(hidden_dims[i - 1])
                self.params['W' + str(i)] = np.random.normal(0, weight_scale, (hidden_dims[i - 2], hidden_dims[i - 1])

                #if the network uses batch normalization
                if self.use_batchnorm == True:
                    self.params['gamma' + str(i)] = np.ones(hidden_dims[i - 1]) #the gammas should be 1
                    self.params['beta' + str(i)] = np.zeros(hidden_dims[i - 1]) #the betas should be zero

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

        out_list = []
        cache_list = []
        dropcache_list = []
        X = X.reshape([X.shape[0], -1])

        for i in np.arange (1, self.num_layers + 1):

            #we will consider three instances:

            if i == 1: #for the layer 1

                #if the network uses batch normalization
                if self.use_batchnorm:
                    #affine -> batchnorm -> relu
                    out1, cache1 = affine_batchnorm_relu_forward(X, self.params['W' + str(i)], self.params['b' + str(i
                                        self.params['gamma' + str(i)], self.params['beta' + str(i)], self.bn_params[i

                else:
                    #affine -> relu
```

```python
                out1, cache1 = affine_relu_forward(X, self.params['W' + str(i)], self.params['b1'])

            cache_list.append(cache1)

            #If dropout is non-zero
            if self.use_dropout:

                #insert a dropout after ReLU layer
                out1, cached = dropout_forward(out1, self.dropout_param)
                dropcache_list.append(cached)

            out_list.append(out1)


        elif i == self.num_layers: #for the last layer

            #do the last layer affine forward
            scores, cachescores = affine_forward(out_list[i - 2], self.params['W' + str(i)], self.params['b' + str

            cache_list.append(cachescores)

        else: #for the layers except the first and the last

            #if the network uses batch normalization
            if self.use_batchnorm:
                #affine -> batchnorm -> relu
                outnow, cachenow = affine_batchnorm_relu_forward(out_list[i - 2], self.params['W' + str(i)], self.
                                                    self.params['gamma' + str(i)], self.params['beta' + str(i)], s

            else:
                #affine -> relu
                outnow, cachenow = affine_relu_forward(out_list[i - 2], self.params['W' + str(i)], self.params['b'

            cache_list.append(cachenow)

            #If dropout is non-zero
            if self.use_dropout:

                #insert a dropout after ReLU layer
                outnow, cached = dropout_forward(outnow, self.dropout_param)
                dropcache_list.append(cached)

            out_list.append(outnow)

    # ================================================================ #
    # 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, dx = softmax_loss(scores, y) #calculate loss and dx by given softmax_loss

    for j in np.arange (self.num_layers, 0, -1): #loop from the last layer to the first layer
        #L2 regularization by multiplying the regularization loss by 0.5(with factor reg)
        loss += (self.reg) * (np.sum(self.params['W' + str(j)] ** 2)) * 0.5

        if j == self.num_layers: #from the last layer
            #backward affine of the second layer
            currentdx, grads['W' + str(j)], grads['b' + str(j)] = affine_backward(dx, cache_list[j - 1])

        else: #for other layers:

            #If dropout is non-zero
            if self.use_dropout:
                currentdx = dropout_backward(currentdx, dropcache_list[j - 1])

            #if the network uses batch normalization
            if self.use_batchnorm:
                currentdx, grads['W' + str(j)], grads['b' + str(j)], grads['gamma' + str(j)], grads['beta' + str(j

            else:
                currentdx, grads['W' + str(j)], grads['b' + str(j)] = affine_relu_backward(currentdx, cache_list[j
```

```
        #update the gradients of W
        grads['W' + str(j)] += (self.reg) * self.params['W' + str(j)]

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

    return loss, grads
```