In [44]:

```python
def batchnorm_forward(x, gamma, beta, bn_param):
    """
    Forward pass for batch normalization.

    During training the sample mean and (uncorrected) sample variance are
    computed from minibatch statistics and used to normalize the incoming data.
    During training we also keep an exponentially decaying running mean of the mean
    and variance of each feature, and these averages are used to normalize data
    at test-time.

    At each timestep we update the running averages for mean and variance using
    an exponential decay based on the momentum parameter:

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    Note that the batch normalization paper suggests a different test-time
    behavior: they compute sample mean and variance for each feature using a
    large number of training images rather than using a running average. For
    this implementation we have chosen to use running averages instead since
    they do not require an additional estimation step; the torch7 implementation
    of batch normalization also uses running averages.

    Input:
    - x: Data of shape (N, D)
    - gamma: Scale parameter of shape (D,)
    - beta: Shift paremeter of shape (D,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: of shape (N, D)
    - cache: A tuple of values needed in the backward pass
    """
    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == 'train':

        # ================================================================ #
        # YOUR CODE HERE:
        #   A few steps here:
        #     (1) Calculate the running mean and variance of the minibatch.
        #     (2) Normalize the activations with the running mean and variance.
        #     (3) Scale and shift the normalized activations.  Store this
        #         as the variable 'out'
        #     (4) Store any variables you may need for the backward pass in
        #         the 'cache' variable.
        # ================================================================ #

        #calculate the sample_mean and sample_var
        sample_mean = np.mean(x, axis = 0)
        sample_var = np.var(x, axis = 0)

        #Calculate the running mean and variance of the minibatch
        running_mean = momentum * running_mean + (1 - momentum) * sample_mean
        running_var = momentum * running_var + (1 - momentum) * sample_var

        #Normalize the activations with the running mean and variance
        x_hat = (x - sample_mean) / (np.sqrt(sample_var + eps))
        #based on the unit activations function

        #Scale and shift the normalized activations
        out = gamma * x_hat + beta

        #Store any variables you may need for the backward pass
        cache = (sample_mean, sample_var, x, x_hat, gamma, eps)



        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    elif mode == 'test':
        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the testing time normalized activation.  Normalize using
        #   the running mean and variance, and then scale and shift appropriately.
        #   Store the output as 'out'.
        # ================================================================ #
```

```python
        out = gamma * ((x - running_mean) / (np.sqrt(running_var + eps))) + beta

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var

    return out, cache

def batchnorm_backward(dout, cache):
    """
    Backward pass for batch normalization.

    For this implementation, you should write out a computation graph for
    batch normalization on paper and propagate gradients backward through
    intermediate nodes.

    Inputs:
    - dout: Upstream derivatives, of shape (N, D)
    - cache: Variable of intermediates from batchnorm_forward.

    Returns a tuple of:
    - dx: Gradient with respect to inputs x, of shape (N, D)
    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    dx, dgamma, dbeta = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
    # ================================================================ #

    #First, we will get the params we need from cache
    sample_mean, sample_var, x, x_hat, gamma, eps = cache
    m = dout.shape[0]

    #Then, calculate the derivatives based on lecture slides
    dldx_hat = dout * gamma #derivative for x_hat

    #derivative for a(in slides)
    dlda = (1 / (np.sqrt(sample_var + eps))) * dldx_hat

    #derivative for mu
    dldmu = (-1 / np.sqrt(sample_var + eps)) * (np.sum(dldx_hat, axis = 0))

    #derivative for e(in slides)
    dlde = (-1 / 2) * (1 / ((sample_var + eps) ** 1.5)) * (x - sample_mean) * dldx_hat

    dldvar = np.sum(dlde, axis = 0) #derivative for variance

    #derivative for x
    dx = dlda + ((2 * (x - sample_mean)) / m) * dldvar + (1 / m) * dldmu

    dgamma = np.sum(dout * x_hat, axis = 0) #derivative for gamma
    dbeta = np.sum(dout, axis = 0) #derivate for beta

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

    return dx, dgamma, dbeta
```

In [ ]:

```python
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
```

```python
                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 - 1])

                else:
                    #affine -> relu
                    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)
```

```python
        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(i)])

            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.params['b
                                                    self.params['gamma' + str(i)], self.params['beta' + str(i)], self.bn_pa

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

            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
    #dxlist = []

    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])
            #dxlist.append(currentdx)

        else: #for other layers:

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

            #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)] = aff
                #dxlist.append(dxcur)

            else:
                currentdx, grads['W' + str(j)], grads['b' + str(j)] = affine_relu_backward(currentdx, cache_list[j - 1])
                #dxlist.append(dxcur)

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

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

    return loss, grads
```