optim.py related code sections

```python
def sgd_momentum(w, dw, config=None):
    """
    Performs stochastic gradient descent with momentum.

    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces to sgd.
    - velocity: A numpy array of the same shape as w and dw used to store a moving
      average of the gradients.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
    v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.

    # ================================================================= #
    # YOUR CODE HERE:
    #   Implement the momentum update formula.  Return the updated weights
    #   as next_w, and the updated velocity as v.
    # ================================================================= #
    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = w + v
    # ================================================================= #
    # END YOUR CODE HERE
    # ================================================================= #

    config['velocity'] = v

    return next_w, config
```

```python
def sgd_nesterov_momentum(w, dw, config=None):
    """
    Performs stochastic gradient descent with Nesterov momentum.

    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces to sgd.
    - velocity: A numpy array of the same shape as w and dw used to store a moving
      average of the gradients.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
    v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the momentum update formula.  Return the updated weights
    #   as next_w, and the updated velocity as v.
    # ================================================================ #
    v_old = v
    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = w + v + config['momentum'] * (v-v_old)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    config['velocity'] = v

    return next_w, config
```

```python
def rmsprop(w, dw, config=None):
    """
    Uses the RMSProp update rule, which uses a moving average of squared gradient
    values to set adaptive per-parameter learning rates.

    config format:
    - learning_rate: Scalar learning rate.
    - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
      gradient cache.
    - epsilon: Small scalar used for smoothing to avoid dividing by zero.
    - beta: Moving average of second moments of gradients.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('decay_rate', 0.99)
    config.setdefault('epsilon', 1e-8)
    config.setdefault('a', np.zeros_like(w))

    next_w = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement RMSProp.  Store the next value of w as next_w.  You need
    #   to also store in config['a'] the moving average of the second
    #   moment gradients, so they can be used for future gradients. Concretely,
    #   config['a'] corresponds to "a" in the lecture notes.
    # ================================================================ #
    config['a'] = config['decay_rate'] * config['a'] + (1 - config['decay_rate']) * dw * dw
    next_w = w - config['learning_rate'] * dw / (np.sqrt(config['a']) + config['epsilon'])
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return next_w, config
```

```python
def adam(w, dw, config=None):
    """
    Uses the Adam update rule, which incorporates moving averages of both the
    gradient and its square and a bias correction term.

    config format:
    - learning_rate: Scalar learning rate.
    - beta1: Decay rate for moving average of first moment of gradient.
    - beta2: Decay rate for moving average of second moment of gradient.
    - epsilon: Small scalar used for smoothing to avoid dividing by zero.
    - m: Moving average of gradient.
    - v: Moving average of squared gradient.
    - t: Iteration number.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-3)
    config.setdefault('beta1', 0.9)
    config.setdefault('beta2', 0.999)
    config.setdefault('epsilon', 1e-8)
    config.setdefault('v', np.zeros_like(w))
    config.setdefault('a', np.zeros_like(w))
    config.setdefault('t', 0)

    next_w = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement Adam.  Store the next value of w as next_w.  You need
    #   to also store in config['a'] the moving average of the second
    #   moment gradients, and in config['v'] the moving average of the
    #   first moments.  Finally, store in config['t'] the increasing time.
    # ================================================================ #
    b1 = config['beta1']
    b2 = config['beta2']
    config['t'] += 1
    config['v'] = b1 * config['v'] + (1-b1) * dw
    config['a'] = b2 * config['a'] + (1-b2) * dw * dw
    v_u = config['v'] / (1-b1**config['t'])
    a_u = config['a'] / (1-b2**config['t'])
    next_w = w - config['learning_rate'] * v_u / (np.sqrt(a_u) + config['epsilon'])
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return next_w, config
```

layers.py related code sections

```python
def affine_forward(x, w, b):
    """
    Computes the forward pass for an affine (fully-connected) layer.

    The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
    examples, where each example x[i] has shape (d_1, ..., d_k). We will
    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
    then transform it to an output vector of dimension M.

    Inputs:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the output of the forward pass.  Notice the dimensions
    #   of w are D x M, which is the transpose of what we did in earlier
    #   assignments.
    # ================================================================ #
    N = x.shape[0]
          # N x D
    out = (x.reshape(N, -1) @ w) + b # N x M
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    cache = (x, w, b)
    return out, cache
```

```python
def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
      - x: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # ================================================================ #

    # dout is N x M
    # dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x M
    # dw should be D x M; it relates to dout through multiplication with x, which is N x D after reshaping
    # db should be M; it is just the sum over dout examples

    N = x.shape[0]
    new_x = x.reshape(N, -1) # N x D
    dx = (dout @ w.T).reshape(x.shape) # N x d1 x ... x dk
    dw = new_x.T @ dout # D x M
    db = np.sum(dout.T, axis=1)  # M * 1

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

    return dx, dw, db
```

```python
def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ================================================================ #
    out = np.maximum(0, x)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    cache = x
    return out, cache

def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLUs).

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    x = cache

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU backward pass
    # ================================================================ #
    dx = (x > 0) * dout
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dx
```

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

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

      sample_mean = np.mean(x, axis=0)
      sample_var = np.var(x, axis=0)

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

      x_hat = (x - sample_mean) / np.sqrt(sample_var + eps)
      out = gamma * x_hat + beta
      cache = sample_mean, sample_var, x_hat, x, 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'.
      # ================================================================ #

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

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

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

 dx, dgamma, dbeta = None, None, None

 # ================================================================ #
 # YOUR CODE HERE:
 #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
 # ================================================================ #
 sample_mean, sample_var, x_hat, x, gamma, eps = cache

 N, D = dout.shape

 dbeta = np.sum(dout, axis=0)
 dgamma = np.sum(dout * x_hat, axis=0)

 dl_dxhat = dout * gamma

 b = 1 / np.sqrt(sample_var + eps)
 a = x - sample_mean
 e = sample_var + eps

 dl_da = dl_dxhat * b
 dl_db = dl_dxhat * a
 dl_de = -0.5 * dl_db / e**(3/2)

 dl_dvar = np.sum(dl_de, axis=0)

 dl_dmu = -b * np.sum(dl_dxhat, axis=0) - dl_dvar * 2 * np.sum(x-sample_mean) / N

 dx = dl_da + 2 * (x-sample_mean) * dl_dvar / N + dl_dmu / N

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

 return dx, dgamma, dbeta
```

```python
def dropout_forward(x, dropout_param):
    """
    Performs the forward pass for (inverted) dropout.

if mode == 'train':
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during training time.
    #   Store the masked and scaled activations in out, and store the
    #   dropout mask as the variable mask.
    # ================================================================ #
    mask = (np.random.rand(*x.shape) < p) / p
    out = x * mask
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

elif mode == 'test':

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during test time.
    # ================================================================ #
    out = x
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
```

```python
def dropout_backward(dout, cache):
    """
    Perform the backward pass for (inverted) dropout.

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (dropout_param, mask) from dropout_forward.
    """
    dropout_param, mask = cache
    mode = dropout_param['mode']

    dx = None
    if mode == 'train':
        # ================================================================ #
        # YOUR CODE HERE:
        #   Implement the inverted dropout backward pass during training time.
        # ================================================================ #
        dx = dout*mask
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    elif mode == 'test':
        # ================================================================ #
        # YOUR CODE HERE:
        #   Implement the inverted dropout backward pass during test time.
        # ================================================================ #
        dx = dout
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    return dx
```

layers_utils.py (implemented affine_batchnorm_relu_forward and affine_batchnorm_relu_backward)

```python
def affine_batchnorm_relu_forward(x, w, b, gamma, beta, bn_param):
    a, fc_cache = affine_forward(x, w, b)
    out, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
    out, relu_cache = relu_forward(out)

    cache = fc_cache, bn_cache, relu_cache
    return out, cache

def affine_batchnorm_relu_backward(dout, cache):
    fc_cache, bn_cache, relu_cache  = cache
    da = relu_backward(dout, relu_cache)
    dx, dgamma, dbeta = batchnorm_backward(da, bn_cache)
    dx, dw, db = affine_backward(dx, fc_cache)
    return dx, dw, db, dgamma, dbeta
```

fc_net.py related code sections

```
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.


  # ================================================================== #
  # 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.
  # ================================================================== #
  dims = [input_dim] + hidden_dims + [num_classes]
  for i in range(self.num_layers):
    if self.use_batchnorm and i < self.num_layers - 1:
      self.params['gamma' + str(i + 1)] = np.ones(dims[i + 1])
      self.params['beta' + str(i + 1)] = np.zeros(dims[i + 1])
    self.params['W' + str(i + 1)] = weight_scale * np.random.randn(dims[i], dims[i + 1])
    self.params['b' + str(i + 1)] = np.zeros(dims[i + 1])

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

```python
def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.


    # ================================================================ #
    # 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.
    # ================================================================ #
    a = {}
    d_cache = {}
    a[0] = [X]
    for i in range(self.num_layers - 1):
        if self.use_batchnorm:
            a[i+1] = affine_batchnorm_relu_forward(a[i][0], self.params['W'+str(i+1)],
                                                            self.params['b'+str(i+1)],
                                                            self.params['gamma'+str(i+1)],
                                                            self.params['beta'+str(i+1)],
                                                            self.bn_params[i])
        else:
            a[i+1] = affine_relu_forward(a[i][0], self.params['W'+str(i+1)], self.params['b'+str(i+1)])

        if self.use_dropout:
            out, d_cache[i+1] = dropout_forward(a[i+1][0], self.dropout_param)
            a[i+1] = out, a[i+1][1]

    # last layer has no relu
    a[self.num_layers] = affine_forward(a[self.num_layers - 1][0], self.params['W' + str(self.num_layers)],
                                        self.params['b' + str(self.num_layers)])
    scores = a[self.num_layers][0]
    scores_cached = a[self.num_layers][1]

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

```
# ================================================================ #
# 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)
for i in range(self.num_layers):
    loss += 0.5 * self.reg * np.sum(self.params['W' + str(i+1)] ** 2)

da = {}
da[self.num_layers], grads['W' + str(self.num_layers)], grads['b' + str(self.num_layers)] = affine_backward(dx, scores_cached)

for i in range(self.num_layers - 1, 0, -1):
    if self.use_dropout:
        da[i+1] = dropout_backward(da[i+1], d_cache[i])
    if self.use_batchnorm:
        da[i], grads['W'+str(i)], grads['b'+str(i)], grads['gamma'+str(i)], grads['beta'+str(i)] = affine_batchnorm_relu_backward(
                                                                                                                      da[i+1], a[i][1])
    else:
        da[i], grads['W'+str(i)], grads['b'+str(i)] = affine_relu_backward(da[i+1], a[i][1])

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

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

return loss, grads
```