```
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names
to be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please
visit
cs231n.stanford.edu.
"""

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)
    """
    out = None
    # ================================================================
#
    # 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.
    # ================================================================
```

```
#

    rx = x.reshape(x.shape[0], -1)
    out = rx.dot(w) + b

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

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


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: 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:
    - 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.
    # Notice:
    #   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
    # ================================================================
#

    rx = x.reshape(x.shape[0], -1)
```

```
    dx = dout.dot(w.T).reshape(x.shape)
    dw = rx.T.dot(dout)
    db = np.sum(dout.T, axis=1)

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

    return dx, dw, db

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 = x * (x > 0)

    # ================================================================
#
    # 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 = dout * (x > 0)

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

    return dx

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

        num_train = x.shape[0]
        sample_mean = (np.sum(x, axis=0)) / num_train
        sample_var = (np.sum((x - sample_mean)**2, axis=0)) /
num_train

        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 = x_hat * gamma + beta

        cache = (x, x_hat, sample_mean, sample_var, eps, gamma, beta)

        #
================================================================= #
        # 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 = x_hat * gamma + 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.
    # ================================================================
#

    x, x_hat, sample_mean, sample_var, eps, gamma, beta = cache
    num_train = x.shape[0]

    dldy = dout
    dldbeta = np.sum(dldy, axis=0)
    dldgamma = np.sum(dldy * x_hat, axis=0)
    dldx_hat = dldy * gamma
    dlda = 1 / np.sqrt(sample_var + eps) * dldx_hat
    dldb = (x - sample_mean) * dldx_hat
    dldc = -1 / (sample_var + eps) * dldb
    dlde = -1 / (2 * (sample_var + eps)**(3/2)) * dldb
```

```
    dldv = np.sum(dlde, axis=0)
    dldmu = -1 / np.sqrt(sample_var + eps) * np.sum(dldx_hat, axis=0)
- dldv * 2 * np.mean(x - sample_mean, axis=0)
    dvdx = 2 / num_train * (x - sample_mean)
    dldx = dlda + dvdx * dldv + 1 / num_train * dldmu

    dx = dldx
    dgamma = dldgamma
    dbeta = dldbeta

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

    return dx, dgamma, dbeta

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

    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
      - p: Dropout parameter. We keep each neuron output with
probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform
dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes
this
        function deterministic, which is needed for gradient checking
but not in
        real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: A tuple (dropout_param, mask). In training mode, mask is
the dropout
      mask that was used to multiply the input; in test mode, mask is
None.
    """
    p, mode = dropout_param['p'], dropout_param['mode']
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])

    mask = None
    out = None
```

```
    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 = mask * x

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

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache

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 = mask * dout

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

def svm_loss(x, y):
    """
    Computes the loss and gradient using for multiclass SVM
classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for
the jth class
      for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for
```

```
x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    N = x.shape[0]
    correct_class_scores = x[np.arange(N), y]
    margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] +
1.0)
    margins[np.arange(N), y] = 0
    loss = np.sum(margins) / N
    num_pos = np.sum(margins > 0, axis=1)
    dx = np.zeros_like(x)
    dx[margins > 0] = 1
    dx[np.arange(N), y] -= num_pos
    dx /= N
    return loss, dx


def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for
the jth class
      for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for
x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """

    probs = np.exp(x - np.max(x, axis=1, keepdims=True))
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```