```python
import numpy as np

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) where x[i] is the ith input.
  We multiply this against a weight matrix of shape (D, M) where
  D = \prod_i d_i

  Inputs:
  x - Input data, of shape (N, d_1, ..., d_k)
  w - Weights, of shape (D, M)
  b - Biases, of shape (M,)

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """
  out = None
  #############################################################################
  # TODO: Implement the affine forward pass. Store the result in out. You     #
  # will need to reshape the input into rows.                                 #
  #############################################################################
  row_dim = x.shape[0]
  col_dim = np.prod(x.shape[1:])
  x_reshape = x.reshape(row_dim, col_dim)
  out = np.dot(x_reshape, w) + b
  #############################################################################
  #                             END OF YOUR CODE                              #
  #############################################################################
  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: 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
  #############################################################################
  # TODO: Implement the affine backward pass.                                 #
  #############################################################################
  row_dim = x.shape[0]
  col_dim = np.prod(x.shape[1:])
  x2 = np.reshape(x, (row_dim, col_dim))

  dx2 = np.dot(dout, w.T) # row_dim x col_dim
  dw = np.dot(x2.T, dout) # col_dim x M
  db = np.dot(dout.T, np.ones(row_dim)) # M x 1

  dx = np.reshape(dx2, x.shape)
  #############################################################################
  #                             END OF YOUR CODE                              #
  #############################################################################
```

```python
  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
  """
  out = None
  ##############################################################################
  # TODO: Implement the ReLU forward pass.                                     #
  ##############################################################################
  # This secures that ReLu never goes below zero
  out = np.maximum(0, x)
  ##############################################################################
  #                              END OF YOUR CODE                              #
  ##############################################################################
  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
  """
  dx, x = None, cache
  ##############################################################################
  # TODO: Implement the ReLU backward pass.                                    #
  ##############################################################################
  # This secures that ReLu never goes below zero
  out = np.maximum(0, x)
  out[out > 0 ] = 1
  dx = out * dout
  ##############################################################################
  #                              END OF YOUR CODE                              #
  ##############################################################################
  return dx

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.
```

```python
    - 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':
      #######################################################################
      # TODO: Implement the training phase forward pass for inverted dropout.   #
      # Store the dropout mask in the mask variable.                            #
      #######################################################################
      preMask = (np.random.rand(*x.shape) < p)
      mask = preMask / p
      out = x * mask
      #######################################################################
      #                              END OF YOUR CODE                           #
      #######################################################################
    elif mode == 'test':
      #######################################################################
      # TODO: Implement the test phase forward pass for inverted dropout.       #
      #######################################################################
      out = x
      #######################################################################
      #                              END OF YOUR CODE                           #
      #######################################################################

    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']
    if mode == 'train':
      #######################################################################
      # TODO: Implement the training phase forward pass for inverted dropout.   #
      # Store the dropout mask in the mask variable.                            #
      #######################################################################
      dx = dout * mask
      #######################################################################
      #                              END OF YOUR CODE                           #
      #######################################################################
    elif mode == 'test':
      dx = dout
    return dx


  def conv_forward_naive(x, w, b, conv_param):
    """
    A naive implementation of the forward pass for a convolutional layer.

    The input consists of N data points, each with C channels, height H and width
    W. We convolve each input with F different filters, where each filter spans
    all C channels and has height HH and width HH.
```

```python

  Input:
  - x: Input data of shape (N, C, H, W)
  - w: Filter weights of shape (F, C, HH, WW)
  - b: Biases, of shape (F,)
  - conv_param: A dictionary with the following keys:
    - 'stride': The number of pixels between adjacent receptive fields in the
      horizontal and vertical directions.
    - 'pad': The number of pixels that will be used to zero-pad the input.

  Returns a tuple of:
  - out: Output data, of shape (N, F, H', W') where H' and W' are given by
    H' = 1 + (H + 2 * pad - HH) / stride
    W' = 1 + (W + 2 * pad - WW) / stride
  - cache: (x, w, b, conv_param)
  """
  out = None
  #############################################################################
  # TODO: Implement the convolutional forward pass.                           #
  # Hint: you can use the function np.pad for padding.                        #
  #############################################################################
  N,C1,H,W = x.shape
  F,C2,HH,WW = w.shape
  stride = conv_param['stride']
  pad = conv_param['pad']

  H_mark = 1 + (H + 2 * pad - HH) / stride
  W_mark = 1 + (W + 2 * pad - WW) / stride
  out = np.zeros((N,F,H_mark,W_mark))

  for n in range(N):
    x_pad = np.pad(x[n,:,:,:], ((0,0),(pad,pad),(pad,pad)), 'constant')
    for f in range(F):
      for h_mark in range(H_mark):
        for w_mark in range(W_mark):
          h1 = h_mark * stride
          h2 = h_mark * stride + HH
          w1 = w_mark * stride
          w2 = w_mark * stride + WW
          window = x_pad[:, h1:h2, w1:w2]
          out[n, f, h_mark, w_mark] = np.sum(window * w[f,:,:,:]) + b[f]
  #############################################################################
  #                             END OF YOUR CODE                              #
  #############################################################################
  cache = (x, w, b, conv_param)
  return out, cache


def conv_backward_naive(dout, cache):
  """
  A naive implementation of the backward pass for a convolutional layer.

  Inputs:
  - dout: Upstream derivatives.
  - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

  Returns a tuple of:
  - dx: Gradient with respect to x
  - dw: Gradient with respect to w
  - db: Gradient with respect to b
  """
  dx, dw, db = None, None, None
  #############################################################################
  # TODO: Implement the convolutional backward pass.                          #
  #############################################################################
  x,w,b,conv_param = cache
  N,C1,H,W = x.shape
```

```python
  F,C2,HH,WW = w.shape
  N,F,H_mark,W_mark = dout.shape

  stride = conv_param['stride']
  pad = conv_param['pad']

  dx = np.zeros(x.shape)
  dw = np.zeros(w.shape)
  db = np.zeros(b.shape)

  for n in xrange(N):
    dx_pad = np.pad(dx[n,:,:,:], ((0,0),(pad,pad),(pad,pad)), 'constant')
    x_pad = np.pad(x[n,:,:,:], ((0,0),(pad,pad),(pad,pad)), 'constant')
    for f in xrange(F):
      for h_mark in xrange(H_mark):
        for w_mark in xrange(W_mark):
          h1 = h_mark * stride
          h2 = h_mark * stride + HH
          w1 = w_mark * stride
          w2 = w_mark * stride + WW
          dx_pad[:, h1:h2, w1:w2] += w[f,:,:,:] * dout[n,f,h_mark,w_mark]
          dw[f,:,:,:] += x_pad[:, h1:h2, w1:w2] * dout[n,f,h_mark,w_mark]
          db[f] += dout[n,f,h_mark,w_mark]
    dx[n,:,:,:] = dx_pad[:,1:-1,1:-1]
  #############################################################################
  #                             END OF YOUR CODE                              #
  #############################################################################
  return dx, dw, db


def max_pool_forward_naive(x, pool_param):
  """
  A naive implementation of the forward pass for a max pooling layer.

  Inputs:
  - x: Input data, of shape (N, C, H, W)
  - pool_param: dictionary with the following keys:
    - 'pool_height': The height of each pooling region
    - 'pool_width': The width of each pooling region
    - 'stride': The distance between adjacent pooling regions

  Returns a tuple of:
  - out: Output data
  - cache: (x, pool_param)
  """
  out = None
  #############################################################################
  # TODO: Implement the max pooling forward pass                              #
  #############################################################################
  N, C, H, W = x.shape
  pool_height = pool_param['pool_height']
  pool_width = pool_param['pool_width']
  stride = pool_param['stride']

  H_mark = 1 + (H - pool_height) / stride
  W_mark = 1 + (W - pool_width) / stride

  out = np.zeros((N, C, H_mark, W_mark))

  for n in xrange(N):
    for h in xrange(H_mark):
      for w in xrange(W_mark):
        h1 = h * stride
        h2 = h * stride + pool_height
        w1 = w * stride
        w2 = w * stride + pool_width
        window = x[n, :, h1:h2, w1:w2]
```

```
            out[n,:,h,w] = np.max(window.reshape((C, pool_height*pool_width)), axis=1)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, pool_param)
    return out, cache


def max_pool_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a max pooling layer.

    Inputs:
    - dout: Upstream derivatives
    - cache: A tuple of (x, pool_param) as in the forward pass.

    Returns:
    - dx: Gradient with respect to x
    """
    dx = None
    ###########################################################################
    # TODO: Implement the max pooling backward pass                           #
    ###########################################################################
    x, pool_param = cache
    N, C, H, W = x.shape

    pool_height = pool_param['pool_height']
    pool_width = pool_param['pool_width']
    stride = pool_param['stride']

    H_mark = 1 + (H - pool_height) / stride
    W_mark = 1 + (W - pool_width) / stride

    dx = np.zeros(x.shape)

    for n in xrange(N):
      for c in xrange(C):
        for h in xrange(H_mark):
          for w in xrange(W_mark):
            h1 = h * stride
            h2 = h * stride + pool_height
            w1 = w * stride
            w2 = w * stride + pool_width

            window = x[n, c, h1:h2, w1:w2]
            window2 = np.reshape(window, (pool_height*pool_width))
            window3 = np.zeros_like(window2)
            window3[np.argmax(window2)] = 1

            dx[n,c,h1:h2,w1:w2] = np.reshape(window3,(pool_height,pool_width)) *
            dout[n,c,h,w]
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    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
```