```
import numpy as np
import pdb




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

  # print("w", w.shape)
  # print("xr", x.shape)
  # print("b", b.shape)
  # print("x shape", x.shape)
  xr = x.reshape(x.shape[0], -1)
  # print("x reshape", xr.shape)

  out = xr.dot(w) + b
  # print("wxr", xr.dot(w).shape) # check this shape to see if h2
shape is even correct???


  # ================================================================ #
  # 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: 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

    # dout: 5 x 3
    # w: 3 x 10
    # x: 10 x 5 which is M x N
    # print("x shape", x.shape)

    # n = 5
    # m =
    # d = 3
    # m = THERE IS SMTH WRONG HERE!!! WITH THE NUNMBERS
    # print("dout", dout.shape)
    # print("w", w.shape)
    dx = np.dot(dout, w.T) #  (N, M) x (M, D) = (N, D) should be D,M
    # print("dout shape", dout.shape)
    # print("w.T shape", w.T.shape)
    # print("x shape", x.shape)
```

```
  # dx = dx.reshape(x.shape)

  # dw = np.dot(x.reshape(x.shape[0], -1).T, dout) # (N, M) x (1 x N)
should be (D, M) original
  xr = x.reshape(x.shape[0], -1)
  dw = np.dot(xr.T, dout) #not sure if this works
  db = np.sum(dout, axis=0) # M i think this one is fine
  # print("x shape", x.shape)
  # print("dx shape", dx.shape, w.shape[0], dout.shape[1])
  # print("dw shape", dw.shape, dout.shape[0],w.shape[0])
  # print("db shape", db.shape, dout.shape[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.
  # ================================================================ #

  f = lambda x: x * (x > 0)
  out = f(x)
  # print("out", out.shape)
  # print("x", x.shape)
  # ================================================================ #
  # 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
  # ================================================================ #

  # ReLU directs linearly to those > 0
  # xr = x.reshape(x.shape[0], -1)
  dx = dout * (x > 0) # hadamard product is element wise operation

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

  return 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
```