```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
```