In [1]:
```
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 t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
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, a
nd
  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]
```

```
  D = w.shape[0]
  x_reshaped = np.reshape(x, (N,D))
  out = x_reshaped.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: 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.
  # ================================================================ #

  #reshape x matrix to be N, D and multiply upstream for the chain rul
e
  N = x.shape[0]
  D = w.shape[0]
  reshaped_x = np.reshape(x, (N, D))
  dw = reshaped_x.T.dot(dout)

  #derivative wrt x
  dx_raw = dout.dot(w.T)
  dx = np.reshape(dx_raw, x.shape)
  #sum derivative for bias
  db = np.sum(dout, axis=0)
```

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

    return dx, dw, db

def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReL
Us).

    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 (Re
LUs).

    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 backward pass multiplies the dout by the indicator function
    #arr[arr > 255] = x
    dx = dout

    #apply indicator. Uses < and not <= because 0 is undefined for ReLU
    dx[x < 0] = 0
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dx

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

    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
```