In [ ]:

```
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))
    x_shape = x.shape
    #Reshaping it as N*D
    #x_shape[0] is equal to N
    x = x.reshape( [ x_shape[0], np.prod( x_shape[1:]) ] )
    out = x.dot(w) + b

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

    cache = (x, w, b, x_shape)
    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, x_shape = 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)
  """
#  print
  dx = np.zeros_like(x)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)

  dx += dout.dot(w.T)
  dw += x.T.dot(dout)
  db += dout.sum( axis = 0)

  # Reshaping dx
  dx = dx.reshape(x_shape)

  # ================================================================ #
  # 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)
  out = np.maximum(x, np.zeros_like(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 batchnorm_forward(x, gamma, beta, bn_param):
  """
  Forward pass for batch normalization.

  During training the sample mean and (uncorrected) sample variance ar
e
  computed from minibatch statistics and used to normalize the incomin
g 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 normali
ze data
  at test-time.

  At each timestep we update the running averages for mean and varianc
e using
  an exponential decay based on the momentum parameter:

  running_mean = momentum * running_mean + (1 - momentum) * sample_mea
```

```
n
  running_var = momentum * running_var + (1 - momentum) * sample_var

  Note that the batch normalization paper suggests a different test-ti
me
  behavior: they compute sample mean and variance for each feature usi
ng a
  large number of training images rather than using a running average.
For
  this implementation we have chosen to use running averages instead s
ince
  they do not require an additional estimation step; the torch7 implem
entation
  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 feature
s
    - running_var Array of shape (D,) giving running variance of featu
res

  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)


#  print("received x: ", x.shape)
  N, D = x.shape
  running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtyp
e))
  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 batch mean and varian
ce.
    #     (3) Scale and shift the normalized activations.  Store this
    #         as the variable 'out'
    #     (4) Store any variables you may need for the backward pass i
n
    #         the 'cache' variable.
    # ================================================================
#
    sample_mean = np.mean(x, axis=0)
    sample_var = np.var(x, axis=0)

    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

    #store in cache
    cache = (mode, x, gamma, sample_mean, sample_var, x_hat, out, eps)

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

  elif mode == 'test':

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the testing time normalized activations.  Normalize
using
    #   the running mean and variance, and then scale and shift approp
riately.
    #   Store the output as 'out'.
    # ================================================================
#

    stddev = np.sqrt(running_var + eps)
    x_hat = (x - running_mean)/stddev
    out = x_hat*gamma + beta

    #store in cache
    cache = (mode, x, gamma, x_hat, out, eps, stddev)
```

```
    # ================================================================
#
    # 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

#  print(out.shape)
  return out, cache

def batchnorm_backward(dout, cache):
  """
  Backward pass for batch normalization.

  For this implementation, you should write out a computation graph fo
r
  batch normalization on paper and propagate gradients backward throug
h
  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
  mode = cache[0]
  # ============================================================== #
  # YOUR CODE HERE:
  #   Implement the batchnorm backward pass, calculating dx, dgamma, a
nd dbeta.
  # ============================================================== #
  if(mode == 'train'):
    mode, x, gamma, sample_mean, sample_var, x_hat, out, eps = cache
#    print(cache)

    N, D = x.shape
```

```
    dl_dbeta = np.sum(dout, axis=0)
#    print(dout.shape, x_hat.shape)
    dl_dgamma = np.sum(dout*x_hat, axis=0)
    dl_dx = dout*gamma


    dl_da = (1/np.sqrt(sample_var + eps))*dl_dx
    dl_du = -(1/np.sqrt(sample_var+eps))*np.sum(dl_dx, axis=0)

    dl_de = -0.5*(1/(sample_var+eps))*(x_hat)*dl_dx

    dl_dvar = np.sum(dl_de, axis=0)

    dl_da = (1/(np.sqrt(sample_var + eps)))*dl_dx

    dx = dl_da + 2*((x-sample_mean)/N)*dl_dvar + (1/N)*dl_du
    dgamma = dl_dgamma
    dbeta = dl_dbeta

  elif(mode == 'test'):
    mode, x, gamma, x_hat, out, eps, stddev = cache
    dl_dbeta = np.sum(dout, axis=0)
    dl_dgamma = np.sum(dout*x_hat, axis=0)
    dx = (gamma*dout)/stddev

  # ================================================================ #
  # 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 drop each neuron output with probabilit
y p.
    - mode: 'test' or 'train'. If the mode is train, then perform drop
out;
      if the mode is test, then just return the input.
    - seed: Seed for the random number generator. Passing seed makes t
his
      function deterministic, which is needed for gradient checking bu
t not in
      real networks.

  Outputs:
```

```
  - out: Array of the same shape as x.
  - cache: A tuple (dropout_param, mask). In training mode, mask is th
e dropout
    mask that was used to multiply the input; in test mode, mask is No
ne.
  """
  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 ti
me.
    #   Store the masked and scaled activations in out, and store the
    #   dropout mask as the variable mask.
    # ================================================================
#
#     print(x.shape)
    mask = (np.random.rand(*x.shape) < (1-p)) / (1-p)
    out = x * mask
    # ================================================================
#
    # 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 t
ime.
    # ==================================================================
#
    dx = dout*mask
    # ==================================================================
#
    # 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 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
  """
  eps = 1e-7
  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] +eps)) / N
  dx = probs.copy()
  dx[np.arange(N), y] -= 1
  dx /= N
  return loss, dx
```