cnn.py

In [ ]:
```
import numpy as np

from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
from nndl.layer_utils import *
from nndl.conv_layer_utils import *

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 to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

class ThreeLayerConvNet(object):
  """
  A three-layer convolutional network with the following architecture:

  conv - relu - 2x2 max pool - affine - relu - affine - softmax

  The network operates on minibatches of data that have shape (N, C, H, W)
  consisting of N images, each with height H and width W and with C input
  channels.
  """

  def __init__(self, input_dim=(3, 32, 32), num_filters=32, filter_size=7,
               hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=0.0,
               dtype=np.float32, use_batchnorm=False):
    """
    Initialize a new network.

    Inputs:
    - input_dim: Tuple (C, H, W) giving size of input data
    - num_filters: Number of filters to use in the convolutional layer
    - filter_size: Size of filters to use in the convolutional layer
    - hidden_dim: Number of units to use in the fully-connected hidden layer
    - num_classes: Number of scores to produce from the final affine layer.
    - weight_scale: Scalar giving standard deviation for random initialization
      of weights.
    - reg: Scalar giving L2 regularization strength
    - dtype: numpy datatype to use for computation.
    """
    self.use_batchnorm = use_batchnorm
    self.params = {}
    self.reg = reg
    self.dtype = dtype


    # ================================================================ #
    # YOUR CODE HERE:
    #   Initialize the weights and biases of a three layer CNN. To initialize:
    #     - the biases should be initialized to zeros.
    #     - the weights should be initialized to a matrix with entries
    #         drawn from a Gaussian distribution with zero mean and
    #         standard deviation given by weight_scale.
    # ================================================================ #
    C, H, W = input_dim

    self.params['W1'] = weight_scale * np.random.randn(num_filters, C, filter_size, filter_size)
    self.params['b1'] = np.zeros(num_filters)

    self.params['W2'] = weight_scale * np.random.randn(num_filters * H * W // 4, hidden_dim)
    self.params['b2'] = np.zeros(hidden_dim)

    self.params['W3'] = weight_scale * np.random.randn(hidden_dim, num_classes)
    self.params['b3'] = np.zeros(num_classes)

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

    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Evaluate loss and gradient for the three-layer convolutional network.
    Input / output: Same API as TwoLayerNet in fc_net.py.
    """
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    W3, b3 = self.params['W3'], self.params['b3']

    # pass conv_param to the forward pass for the convolutional layer
    filter_size = W1.shape[2]
    conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}

    # pass pool_param to the forward pass for the max-pooling layer
    pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

    scores = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the three layer CNN.  Store the output
    #   scores as the variable "scores".
    # ================================================================ #

    h1, cache1 = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
    h2, cache2 = affine_relu_forward(h1, W2, b2)
    scores, cache3 = affine_forward(h2, W3, b3)

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

    if y is None:
      return scores

    loss, grads = 0, {}
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of the three layer CNN.  Store the grads
    #   in the grads dictionary, exactly as before (i.e., the gradient of
    #   self.params[k] will be grads[k]).  Store the loss as "loss", and
    #   don't forget to add regularization on ALL weight matrices.
    # ================================================================ #

    loss, dscores = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2) + np.sum(W3**2))

    dx3, dW3, db3 = affine_backward(dscores, cache3)
    dW3 += self.reg * W3

    dx2, dW2, db2 = affine_relu_backward(dx3, cache2)
    dW2 += self.reg * W2

    dx1, dW1, db1 = conv_relu_pool_backward(dx2, cache1)
    dW1 += self.reg * W1

    grads['W1'], grads['b1'] = dW1, db1
    grads['W2'], grads['b2'] = dW2, db2
    grads['W3'], grads['b3'] = dW3, db3

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

    return loss, grads


pass
```

conv_layers.py

In [ ]:
```
import numpy as np
from nndl.layers import *
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 to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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.

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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of a convolutional neural network.
    #   Store the output as 'out'.
    #   Hint: to pad the array, you can use the function np.pad.
    # ================================================================ #

    N, C, H, W = x.shape
    F, _, HH, WW = w.shape

    H_out = 1 + (H + 2 * pad - HH) // stride
    W_out = 1 + (W + 2 * pad - WW) // stride

    x_padded = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode='constant', constant_values=0)

    out = np.zeros((N, F, H_out, W_out))

    for n in range(N):  # all input images
        for f in range(F):  # all filters
            for i in range(H_out):
                for j in range(W_out):
                    h_start = i * stride
                    h_end = h_start + HH
                    w_start = j * stride
                    w_end = w_start + WW
                    out[n, f, i, j] = np.sum(x_padded[n, :, h_start:h_end, w_start:w_end] * w[f]) + b[f]

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

    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

    N, F, out_height, out_width = dout.shape
    x, w, b, conv_param = cache

    stride, pad = [conv_param['stride'], conv_param['pad']]
    xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
    num_filts, _, f_height, f_width = w.shape

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of a convolutional neural network.
    #   Calculate the gradients: dx, dw, and db.
    # ================================================================ #

    dx = np.zeros_like(x)
    dw = np.zeros_like(w)
    db = np.zeros_like(b)
```

```
  dx_pad = np.zeros_like(xpad)

  db = np.sum(dout, axis=(0, 2, 3))

  for f in range(num_filts):
      for i in range(out_height):
          for j in range(out_width):
              h_start = i * stride
              h_end = h_start + f_height
              w_start = j * stride
              w_end = w_start + f_width
              dw[f] += np.sum(xpad[:, :, h_start:h_end, w_start:w_end] * dout[:, f, i, j][:, None, None, None], axis=0)

  for n in range(N):
      for f in range(F):
          for i in range(out_height):
              for j in range(out_width):
                  h_start = i * stride
                  h_end = h_start + f_height
                  w_start = j * stride
                  w_end = w_start + f_width
                  dx_pad[n, :, h_start:h_end, w_start:w_end] += w[f] * dout[n, f, i, j]

  dx = dx_pad[:, :, pad:-pad, pad:-pad] if pad > 0 else dx_pad

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

  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

  # ================================================================ #
  # YOUR CODE HERE:
  #   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_out = 1 + (H - pool_height) // stride
  W_out = 1 + (W - pool_width) // stride

  out = np.zeros((N, C, H_out, W_out))

  for n in range(N):
      for c in range(C):
          for i in range(H_out):
              for j in range(W_out):
                  h_start = i * stride
                  h_end = h_start + pool_height
                  w_start = j * stride
                  w_end = w_start + pool_width
                  out[n, c, i, j] = np.max(x[n, c, h_start:h_end, w_start:w_end])



  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #
  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
  x, pool_param = cache
  pool_height, pool_width, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the max pooling backward pass.
  # ================================================================ #

  N, C, H, W = x.shape
  H_out = dout.shape[2]
  W_out = dout.shape[3]

  dx = np.zeros_like(x)

  for n in range(N):
      for c in range(C):
          for i in range(H_out):
              for j in range(W_out):
                  h_start = i * stride
                  h_end = h_start + pool_height
                  w_start = j * stride
                  w_end = w_start + pool_width
                  x_pool_region = x[n, c, h_start:h_end, w_start:w_end]

                  max_val = np.max(x_pool_region)

                  mask = (x_pool_region == max_val)

                  dx[n, c, h_start:h_end, w_start:w_end] += dout[n, c, i, j] * mask


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

  return dx

def spatial_batchnorm_forward(x, gamma, beta, bn_param):
  """
  Computes the forward pass for spatial batch normalization.

  Inputs:
  - x: Input data of shape (N, C, H, W)
  - gamma: Scale parameter, of shape (C,)
  - beta: Shift parameter, of shape (C,)
  - bn_param: Dictionary with the following keys:
    - mode: 'train' or 'test'; required
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance. momentum=0 means that
      old information is discarded completely at every time step, while
      momentum=1 means that new information is never incorporated. The
      default of momentum=0.9 should work well in most situations.
    - running_mean: Array of shape (D,) giving running mean of features
    - running_var Array of shape (D,) giving running variance of features

  Returns a tuple of:
  - out: Output data, of shape (N, C, H, W)
  - cache: Values needed for the backward pass
  """
  out, cache = None, None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the spatial batchnorm forward pass.
  #
  #   You may find it useful to use the batchnorm forward pass you
  #   implemented in HW #4.
  # ================================================================ #

  N, C, H, W = x.shape
  mode = bn_param['mode']
  eps = bn_param.get('eps', 1e-5)
  momentum = bn_param.get('momentum', 0.9)
  cache = {}

  running_mean = bn_param.get('running_mean', np.zeros(C, dtype=x.dtype))
  running_var = bn_param.get('running_var', np.zeros(C, dtype=x.dtype))

  if(mode =='train'):
    sample_mean = np.mean(x, axis=(0, 2, 3))
    sample_var = np.var(x, axis=(0, 2, 3))
    x_hat = (x - sample_mean.reshape(1, C, 1, 1)) / np.sqrt(sample_var.reshape(1, C, 1, 1) + eps)
    out = gamma.reshape(1, C, 1, 1) * x_hat + beta.reshape(1, C, 1, 1)

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    cache = (x, x_hat, sample_mean, sample_var, gamma, beta, eps)

  elif(mode == 'test'):
    x_hat = (x - running_mean.reshape(1, C, 1, 1)) / np.sqrt(running_var.reshape(1, C, 1, 1) + eps)
    out = gamma.reshape(1, C, 1, 1) * x_hat + beta.reshape(1, C, 1, 1)

  else:
    raise ValueError('Invalid forward-batchnorm mode %s' % mode)

  bn_param['running_mean'] = running_mean
  bn_param['running_var'] = running_var

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

  return out, cache


def spatial_batchnorm_backward(dout, cache):
  """
  Computes the backward pass for spatial batch normalization.

  Inputs:
  - dout: Upstream derivatives, of shape (N, C, H, W)
  - cache: Values from the forward pass

  Returns a tuple of:
  - dx: Gradient with respect to inputs, of shape (N, C, H, W)
  - dgamma: Gradient with respect to scale parameter, of shape (C,)
  - dbeta: Gradient with respect to shift parameter, of shape (C,)
  """
  dx, dgamma, dbeta = None, None, None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the spatial batchnorm backward pass.
  #
  #   You may find it useful to use the batchnorm forward pass you
  #   implemented in HW #4.
  # ================================================================ #

  N, C, H, W = dout.shape
  x, x_hat, sample_mean, sample_var, gamma, beta, eps = cache
  V = N * H * W

  dbeta = np.sum(dout, axis=(0, 2, 3))
  dgamma = np.sum(dout * x_hat, axis=(0, 2, 3))
  dx_hat = dout * gamma.reshape(1, C, 1, 1)

  dsample_var = np.sum(dx_hat * (x - sample_mean.reshape(1, C, 1, 1)) * (-0.5) * (sample_var.reshape(1, C, 1, 1) + eps)**(-1.5), axis=(0, 2, 3))
  dsample_mean = np.sum(dx_hat * (-1) / np.sqrt(sample_var.reshape(1, C, 1, 1) + eps), axis=(0, 2, 3)) + dsample_var * np.mean(-2 * (x - sample_m

  dx = dx_hat / np.sqrt(sample_var.reshape(1, C, 1, 1) + eps) + dsample_var.reshape(1, C, 1, 1) * 2 * (x - sample_mean.reshape(1, C, 1, 1)) / V +


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

  return dx, dgamma, dbeta
```

layers.py

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 to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
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, 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.
    # ================================================================ #

    x_reshape = x.reshape(x.shape[0], -1)
    out = np.dot(x_reshape, 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.
    # ================================================================ #

    x_reshaped = x.reshape(x.shape[0], -1)

    db = np.sum(dout, axis=0)
    dw = np.dot(x_reshaped.T, dout)
    dx = np.dot(dout, w.T).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 (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.
    # ================================================================ #

    relu = lambda x: x * (x > 0)
    out = relu(x)

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

```python
  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
  # ================================================================ #

  dx = dout * (x > 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 are
  computed from minibatch statistics and used to normalize the incoming 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 normalize data
  at test-time.

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

  running_mean = momentum * running_mean + (1 - momentum) * sample_mean
  running_var = momentum * running_var + (1 - momentum) * sample_var

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

  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)

  N, D = x.shape
  running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
  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 sample mean and variance.
    #     (3) Scale and shift the normalized activations.  Store this
    #         as the variable 'out'
    #     (4) Store any variables you may need for the backward pass in
    #         the 'cache' variable.
    # ================================================================ #

    sample_mean = np.mean(x, axis=0)
    sample_var = np.var(x, axis=0)

    # Normalize the input
    x_hat = (x - sample_mean) / np.sqrt(sample_var + eps)

    # Scale and shift
    out = gamma * x_hat + beta

    # Update running averages
    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    cache = (x, x_hat, sample_mean, sample_var, gamma, beta, eps)

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

  elif mode == 'test':

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

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

    # ================================================================ #
    # 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

  return out, cache

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

  For this implementation, you should write out a computation graph for
  batch normalization on paper and propagate gradients backward through
  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

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
  # ================================================================ #

  x, x_hat, sample_mean, sample_var, gamma, beta, eps = cache
  N, D = x.shape

  dbeta = np.sum(dout, axis=0)
  dgamma = np.sum(dout * x_hat, axis=0)
  dx_hat = dout * gamma

  # gradient of variance
  dvar = np.sum(dx_hat * (x - sample_mean) * -0.5 * (sample_var + eps) ** (-1.5), axis=0)

  # gradient of mean
  dmean = np.sum(dx_hat * -1 / np.sqrt(sample_var + eps), axis=0) + dvar * np.sum(-2 * (x - sample_mean) / N, axis=0)

  # gradient of x
  dx = dx_hat / np.sqrt(sample_var + eps) + dvar * 2 * (x - sample_mean) / N + dmean / N

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

```python
  # ================================================================ #

  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 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.
  - 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':
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the dropout forward pass during training time.
    #   Store the masked and scaled activations in out, and store the
    #   dropout mask as the variable mask.
    # ================================================================ #

    # Inverted dropout
    mask = (np.random.rand(*x.shape) < p) / p
    out = x * mask

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

  elif mode == 'test':

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the 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 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 dropout backward pass during training time.
    # ================================================================ #

    dx = dout * mask

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
  elif mode == 'test':
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the 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 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
```

optim.py

In [ ]:
```
import numpy as np

"""
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 to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

"""
This file implements various first-order update rules that are commonly used for
training neural networks. Each update rule accepts current weights and the
gradient of the loss with respect to those weights and produces the next set of
weights. Each update rule has the same interface:

def update(w, dw, config=None):

Inputs:
  - w: A numpy array giving the current weights.
  - dw: A numpy array of the same shape as w giving the gradient of the
    loss with respect to w.
```

```python
  - config: A dictionary containing hyperparameter values such as learning rate,
    momentum, etc. If the update rule requires caching values over many
    iterations, then config will also hold these cached values.

Returns:
  - next_w: The next point after the update.
  - config: The config dictionary to be passed to the next iteration of the
    update rule.

NOTE: For most update rules, the default learning rate will probably not perform
well; however the default values of the other hyperparameters should work well
for a variety of different problems.

For efficiency, update rules may perform in-place updates, mutating w and
setting next_w equal to w.
"""


def sgd(w, dw, config=None):
  """
  Performs vanilla stochastic gradient descent.

  config format:
  - learning_rate: Scalar learning rate.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)

  w -= config['learning_rate'] * dw
  return w, config


def sgd_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to store a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
  v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.

  # =============================================================== #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weights
  #   as next_w, and the updated velocity as v.
  # =============================================================== #

  v = config['momentum'] * v - config['learning_rate'] * dw
  next_w = w + v

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

  config['velocity'] = v

  return next_w, config

def sgd_nesterov_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with Nesterov momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to store a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
  v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.

  # =============================================================== #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weights
  #   as next_w, and the updated velocity as v.
  # =============================================================== #

  mu = config['momentum']
  learning_rate = config['learning_rate']

  v_old = v
  v = mu * v - learning_rate * dw
  next_w = w + v + mu * (v - v_old)

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

  config['velocity'] = v

  return next_w, config

def rmsprop(w, dw, config=None):
  """
  Uses the RMSProp update rule, which uses a moving average of squared gradient
  values to set adaptive per-parameter learning rates.

  config format:
  - learning_rate: Scalar learning rate.
  - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
    gradient cache.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero.
  - beta: Moving average of second moments of gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('decay_rate', 0.99)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('a', np.zeros_like(w))

  next_w = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement RMSProp.  Store the next value of w as next_w.  You need
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, so they can be used for future gradients. Concretely,
  #   config['a'] corresponds to "a" in the lecture notes.
  # ================================================================ #

  decay_rate = config['decay_rate']
  epsilon = config['epsilon']
  learning_rate = config['learning_rate']

  config['a'] = decay_rate * config['a'] + (1.0 - decay_rate) * dw * dw
  next_w = w - (learning_rate / (np.sqrt(config['a']) + epsilon)) * dw

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

  return next_w, config


def adam(w, dw, config=None):
  """
  Uses the Adam update rule, which incorporates moving averages of both the
  gradient and its square and a bias correction term.

  config format:
  - learning_rate: Scalar learning rate.
  - beta1: Decay rate for moving average of first moment of gradient.
  - beta2: Decay rate for moving average of second moment of gradient.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero.
  - m: Moving average of gradient.
  - v: Moving average of squared gradient.
  - t: Iteration number.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-3)
  config.setdefault('beta1', 0.9)
  config.setdefault('beta2', 0.999)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('v', np.zeros_like(w))
  config.setdefault('a', np.zeros_like(w))
  config.setdefault('t', 0)

  next_w = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement Adam.  Store the next value of w as next_w.  You need
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, and in config['v'] the moving average of the
  #   first moments.  Finally, store in config['t'] the increasing time.
  # ================================================================ #
```

```
    config['v'] = config['beta1'] * config['v'] + (1 - config['beta1']) * dw
    config['a'] = config['beta2'] * config['a'] + (1 - config['beta2']) * dw * dw
    config['t'] += 1

    v_u = config['v'] / (1 - np.power(config['beta1'], config['t']))
    a_u = config['a'] / (1 - np.power(config['beta2'], config['t']))

    next_w = w - config['learning_rate'] / (np.sqrt(a_u) + config['epsilon']) * v_u
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return next_w, config
```