py_files

February 19, 2024

0.1 fc_net.py

[ ]:
```
import numpy as np
import pdb

from .layers import *
from .layer_utils import *

"""
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 TwoLayerNet(object):
  """
  A two-layer fully-connected neural network with ReLU nonlinearity and
  softmax loss that uses a modular layer design. We assume an input dimension
  of D, a hidden dimension of H, and perform classification over C classes.

  The architecure should be affine - relu - affine - softmax.

  Note that this class does not implement gradient descent; instead, it
  will interact with a separate Solver object that is responsible for running
  optimization.

  The learnable parameters of the model are stored in the dictionary
  self.params that maps parameter names to numpy arrays.
  """

  def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
               dropout=0, weight_scale=1e-3, reg=0.0):
    """
    Initialize a new network.
```

```
    Inputs:
    - input_dim: An integer giving the size of the input
    - hidden_dims: An integer giving the size of the hidden layer
    - num_classes: An integer giving the number of classes to classify
    - dropout: Scalar between 0 and 1 giving dropout strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - reg: Scalar giving L2 regularization strength.
    """
    self.params = {}
    self.reg = reg

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ============================================================ #

    self.params['W1'] = np.random.normal(0, weight_scale, (input_dim,␣
↪hidden_dims))
    self.params['W2'] = np.random.normal(0, weight_scale, (hidden_dims,␣
↪num_classes))
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['b2'] = np.zeros(num_classes)

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

  def loss(self, X, y=None):
    """
    Compute loss and gradient for a minibatch of data.

    Inputs:
    - X: Array of input data of shape (N, d_1, ..., d_k)
    - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

    Returns:
    If y is None, then run a test-time forward pass of the model and return:
    - scores: Array of shape (N, C) giving classification scores, where
      scores[i, c] is the classification score for X[i] and class c.
```

```
    If y is not None, then run a training-time forward and backward pass and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping parameter
      names to gradients of the loss with respect to those parameters.
    """
    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ============================================================ #

    h, cacheh = affine_relu_forward(X, self.params['W1'], self.params['b1'])
    z, cachez = affine_forward(h, self.params['W2'], self.params['b2'])
    scores = z

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

    # If y is None then we are in test mode so just return scores
    if y is None:
      return scores

    loss, grads = 0, {}
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of the two-layer neural net.  Store
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
    #   i.e., grads[k] holds the gradient for self.params[k].
    #
    #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ============================================================ #

    loss, grad = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(self.params['W1']**2) + np.sum(self.
↪params['W2']**2))
```

```
    dh, dw2, db2 = affine_backward(grad, cachez)
    dx, dw1, db1 = affine_relu_backward(dh, cacheh)

    grads['W1'] = dw1 + (self.reg * self.params['W1'])
    grads['W2'] = dw2 + (self.reg * self.params['W2'])
    grads['b1'] = db1
    grads['b2'] = db2

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

    return loss, grads


class FullyConnectedNet(object):
    """
    A fully-connected neural network with an arbitrary number of hidden layers,
    ReLU nonlinearities, and a softmax loss function. This will also implement
    dropout and batch normalization as options. For a network with L layers,
    the architecture will be

    {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

    where batch normalization and dropout are optional, and the {...} block is
    repeated L - 1 times.

    Similar to the TwoLayerNet above, learnable parameters are stored in the
    self.params dictionary and will be learned using the Solver class.
    """

    def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
                 dropout=0, use_batchnorm=False, reg=0.0,
                 weight_scale=1e-2, dtype=np.float32, seed=None):
        """
        Initialize a new FullyConnectedNet.

        Inputs:
            - hidden_dims: A list of integers giving the size of each hidden␣
 ↪layer.
        - input_dim: An integer giving the size of the input.
        - num_classes: An integer giving the number of classes to classify.
        - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0␣
 ↪then
          the network should not use dropout at all.
        - use_batchnorm: Whether or not the network should use batch␣
 ↪normalization.
```

```
        - reg: Scalar giving L2 regularization strength.
        - weight_scale: Scalar giving the standard deviation for random
          initialization of the weights.
        - dtype: A numpy datatype object; all computations will be performed␣
↪using
          this datatype. float32 is faster but less accurate, so you should use
          float64 for numeric gradient checking.
        - seed: If not None, then pass this random seed to the dropout layers.␣
↪This
          will make the dropout layers deteriminstic so we can gradient check␣
↪the
          model.
        """
        self.use_batchnorm = use_batchnorm
        self.use_dropout = dropout > 0
        self.reg = reg
        self.num_layers = 1 + len(hidden_dims)
        self.dtype = dtype
        self.params = {}

        # ============================================================ #
        # YOUR CODE HERE:
        #   Initialize all parameters of the network in the self.params␣
↪dictionary.
        #   The weights and biases of layer 1 are W1 and b1; and in general the
        #   weights and biases of layer i are Wi and bi. The
        #   biases are initialized to zero and the weights are initialized
        #   so that each parameter has mean 0 and standard deviation␣
↪weight_scale.
        #
        #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
        #   parameters to zero.  The gamma and beta parameters for layer 1␣
↪should
        #   be self.params['gamma1'] and self.params['beta1'].  For layer 2,␣
↪they
        #   should be gamma2 and beta2, etc. Only use batchnorm if self.
↪use_batchnorm
        #   is true and DO NOT do batch normalize the output scores.
        # ============================================================ #

        for i in range(0, self.num_layers):
            if i == 0:
                self.params['W' + str(i + 1)] = np.random.normal(0,␣
↪weight_scale, (input_dim, hidden_dims[i]))
                self.params['b' + str(i + 1)] = np.zeros(hidden_dims[i])
                if self.use_batchnorm:
```

```
                self.params['gamma' + str(i + 1)] = np.ones(hidden_dims[i])
                self.params['beta' + str(i + 1)] = np.zeros(hidden_dims[i])
        elif i == self.num_layers - 1:
            self.params['W' + str(i + 1)] = np.random.normal(0,␣
↪weight_scale, (hidden_dims[i - 1], num_classes))
            self.params['b' + str(i + 1)] = np.zeros(num_classes)
        else:
            self.params['W' + str(i + 1)] = np.random.normal(0,␣
↪weight_scale, (hidden_dims[i - 1], hidden_dims[i]))
            self.params['b' + str(i + 1)] = np.zeros(hidden_dims[i])
            if self.use_batchnorm:
                self.params['gamma' + str(i + 1)] = np.ones(hidden_dims[i])
                self.params['beta' + str(i + 1)] = np.zeros(hidden_dims[i])


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

        # When using dropout we need to pass a dropout_param dictionary to each
        # dropout layer so that the layer knows the dropout probability and the␣
↪mode
        # (train / test). You can pass the same dropout_param to each dropout␣
↪layer.
        self.dropout_param = {}
        if self.use_dropout:
            self.dropout_param = {'mode': 'train', 'p': dropout}
            if seed is not None:
                self.dropout_param['seed'] = seed

        # With batch normalization we need to keep track of running means and
        # variances, so we need to pass a special bn_param object to each batch
        # normalization layer. You should pass self.bn_params[0] to the forward␣
↪pass
        # of the first batch normalization layer, self.bn_params[1] to the␣
↪forward
        # pass of the second batch normalization layer, etc.
        self.bn_params = []
        if self.use_batchnorm:
            self.bn_params = [{'mode': 'train'} for i in np.arange(self.
↪num_layers - 1)]

        # Cast all parameters to the correct datatype
        for k, v in self.params.items():
            self.params[k] = v.astype(dtype)
```

```
  def loss(self, X, y=None):
      """
      Compute loss and gradient for the fully-connected net.

      Input / output: Same as TwoLayerNet above.
      """
      X = X.astype(self.dtype)
      mode = 'test' if y is None else 'train'

      # Set train/test mode for batchnorm params and dropout param since they
      # behave differently during training and testing.
      if self.dropout_param is not None:
          self.dropout_param['mode'] = mode
      if self.use_batchnorm:
          for bn_param in self.bn_params:
              bn_param[mode] = mode

      scores = None

      # ============================================================ #
      # YOUR CODE HERE:
      #   Implement the forward pass of the FC net and store the output
      #   scores as the variable "scores".
      #
      #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
      #   between the affine_forward and relu_forward layers.  You may
      #   also write an affine_batchnorm_relu() function in layer_utils.py.
      #
      #   DROPOUT: If dropout is non-zero, insert a dropout layer after
      #   every ReLU layer.
      # ============================================================ #

      H = []
      cacheH = []
      cache_dropout = []
      for i in range(0, self.num_layers):
          W = self.params['W' + str(i + 1)]
          b = self.params['b' + str(i + 1)]
          if i == 0:
              if self.use_batchnorm:
                  gamma = self.params['gamma' + str(i + 1)]
                  beta = self.params['beta' + str(i + 1)]
                  h, cache = affine_batchnorm_forward(X, W, b, gamma, beta,␣
↪self.bn_params[i])
              else:
                  h, cache = affine_relu_forward(X, W, b)
```

```
                if self.use_dropout:
                    h, dropout_cache = dropout_forward(h, self.dropout_param)
                    cache_dropout.append(dropout_cache)

                H.append(h)
                cacheH.append(cache)
            elif i == self.num_layers - 1:
                scores, cache =  affine_forward(H[i - 1], W, b)
                cacheH.append(cache)
            else:
                if self.use_batchnorm:
                    gamma = self.params['gamma' + str(i + 1)]
                    beta = self.params['beta' + str(i + 1)]
                    h, cache = affine_batchnorm_forward(H[i - 1], W, b, gamma,␣
↪beta, self.bn_params[i])
                else:
                    h, cache = affine_relu_forward(H[i - 1], W, b)

                if self.use_dropout:
                    h, dropout_cache = dropout_forward(h, self.dropout_param)
                    cache_dropout.append(dropout_cache)

                H.append(h)
                cacheH.append(cache)

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

        # If test mode return early
        if mode == 'test':
            return scores

        loss, grads = 0.0, {}
        # ============================================================ #
        # YOUR CODE HERE:
        #   Implement the backwards pass of the FC net and store the gradients
        #   in the grads dict, so that grads[k] is the gradient of self.
↪params[k]
        #   Be sure your L2 regularization includes a 0.5 factor.
        #
        #   BATCHNORM: Incorporate the backward pass of the batchnorm.
        #
        #   DROPOUT: Incorporate the backward pass of dropout.
        # ============================================================ #
```

```
        loss, grad = softmax_loss(scores, y)

        for i in reversed(range(self.num_layers)):
            loss += (0.5 * self.reg * np.sum(self.params['W' + str(i + 1)]**2))

            if i == self.num_layers - 1:
                grad, grads['W' + str(i + 1)], grads['b' + str(i + 1)] =␣
↪affine_backward(grad, cacheH[i])
            else:
                if self.use_dropout:
                    grad = dropout_backward(grad, cache_dropout[i])

                if self.use_batchnorm:
                    grad, grads['W' + str(i + 1)], grads['b' + str(i + 1)],␣
↪grads['gamma' + str(i + 1)], grads['beta' + str(i + 1)] =␣
↪affine_batchnorm_backward(grad, cacheH[i])
                else:
                    grad, grads['W' + str(i + 1)], grads['b' + str(i + 1)] =␣
↪affine_relu_backward(grad, cacheH[i])


            grads['W' + str(i + 1)] += self.reg * self.params['W' + str(i + 1)]

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

        return loss, grads
```

0.2 layer_utils.py

[ ]:
```
from .layers import *

"""
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_relu_forward(x, w, b):
  """
  Convenience layer that performs an affine transform followed by a ReLU
```

```
  Inputs:
  - x: Input to the affine layer
  - w, b: Weights for the affine layer

  Returns a tuple of:
  - out: Output from the ReLU
  - cache: Object to give to the backward pass
  """
  a, fc_cache = affine_forward(x, w, b)
  out, relu_cache = relu_forward(a)
  cache = (fc_cache, relu_cache)
  return out, cache


def affine_relu_backward(dout, cache):
  """
  Backward pass for the affine-relu convenience layer
  """
  fc_cache, relu_cache = cache
  da = relu_backward(dout, relu_cache)
  dx, dw, db = affine_backward(da, fc_cache)
  return dx, dw, db


def affine_batchnorm_forward(x, w, b, gamma, beta, bn_param):
  """
  Convenience layer that performs an affine transform followed by a batchorm␣
 ↪then ReLu

  Inputs:
  - x: Input to the affine layer
  - w, b: Weights for the affine layer

  Returns a tuple of:
  - out: Output from the ReLU
  - cache: Object to give to the backward pass
  """
  a, fc_cache = affine_forward(x, w, b)
  bn, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
  out, relu_cache = relu_forward(bn)
  cache = (fc_cache, relu_cache, bn_cache)
  return out, cache

def affine_batchnorm_backward(dout, cache):
  """
  Backward pass for the affine-batchnorm-relu convenience layer
```

```
  """
  fc_cache, relu_cache, bn_cache = cache
  da = relu_backward(dout, relu_cache)
  dh, dgamma, dbeta = batchnorm_backward(da, bn_cache)
  dx, dw, db = affine_backward(dh, fc_cache)
  return dx, dw, db, dgamma, dbeta
```

0.3 layers.py

[ ]:
```
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.
    # ============================================================ #
```

```
    out = x.reshape(x.shape[0], -1).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: 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:
        - 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.
    # Notice:
    #   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
    # ============================================================ #

    dx = dout.dot(w.T).reshape(x.shape)
    dw = x.reshape(x.shape[0], -1).T.dot(dout)
    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 (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.
    # ================================================================ #

    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 (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
    cache = {}
    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.
        # ============================================================ #

        mu = x.mean(axis=0)
        var = np.var(x, axis=0)
        xhat = (x - mu) / np.sqrt(var + eps)
        running_mean = momentum * running_mean + (1 - momentum) * mu
        running_var = momentum * running_var + (1 - momentum) * var
        out = gamma * xhat + beta

        cache['mu'] = mu
        cache['sub'] = x - mu
        cache['xhat'] = xhat
        cache['sqrt'] = np.sqrt(var + eps)
        cache['gamma'] = gamma

        # ============================================================ #
        # 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'.
        # ================================================================ #

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

    N = dout.shape[0]
    xhat = cache['xhat']
```

```
    sub = cache['sub']
    gamma = cache['gamma']
    sqrt = cache['sqrt']

    dgamma = np.sum(dout * xhat, axis=0)
    dbeta = np.sum(dout, axis=0)

    dxhat = dout * gamma
    dvar = (-0.5 * np.sum(dxhat * sub, axis=0)) / sqrt**3
    dsum = dxhat / sqrt
    dmu = ((2 / N) * sub * dvar * np.ones_like(dout))
    dvarx = -np.sum(dmu + dsum, axis=0) * np.ones_like(dout) / N
    dx = dsum + dmu + dvarx

    # ============================================================== #
    # 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 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 inverted dropout forward pass during training time.
        #   Store the masked and scaled activations in out, and store the
        #   dropout mask as the variable mask.
        # ============================================================ #

        mask = (np.random.random_sample(x.shape) < p) / 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 time.
        # ================================================================ #

        dx = mask * dout

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

0.4 optim.py

[ ]:
```
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.
  - 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.
    # ================================================================ #

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

    v_new = (momentum * v) - (learning_rate * dw)
    next_w = w + v_new + (momentum * (v_new - v))
    v = v_new

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

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

    config['a'] = (decay_rate * a) + ((1 - decay_rate) * dw**2)
    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.
    # ================================================================ #

    learning_rate = config['learning_rate']
    beta1 = config['beta1']
    beta2 = config['beta2']
    epsilon = config['epsilon']
    v = config['v']
    a = config['a']

    config['t'] += 1
    config['v'] = (beta1 * v) + ((1 - beta1) * dw)
    config['a'] = (beta2 * a) + ((1 - beta2) * dw**2)
    v_tilde = (1 / (1 - (beta1**config['t']))) * config['v']
    a_tilde = (1 / (1 - (beta2**config['t']))) * config['a']
    next_w = w - ((learning_rate / (np.sqrt(a_tilde) - epsilon)) * v_tilde)

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

    return next_w, config
```