```python
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_reshaped = x.reshape(x.shape[0], -1)
  out = X_reshaped @ 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
  # ================================================================ #
  X_reshaped = x.reshape(x.shape[0], -1)
  dx = dout @ w.T
  dx = dx.reshape(x.shape)  # Reshape back to (N, d1, ..., dk)
  dw = X_reshaped.T @ 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) # Indicator Function

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

  return dx
```

```python
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)  # Mean across batch
    sample_var = np.var(x, axis=0)  # Variance across batch

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

    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
    cache = None


    # ============================================================ #
    # 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, mean, var, gamma, beta, eps = cache
  N, D = x.shape

  # Gradients w.r.t. beta and gamma
  dbeta = np.sum(dout, axis=0)
  dgamma = np.sum(dout * x_hat, axis=0)

  # Gradients w.r.t. x
  dx_hat = dout * gamma
  dvar = np.sum(dx_hat * (x - mean) * -0.5 * (var + eps) ** (-1.5), axis=0)
  dmean = np.sum(dx_hat * -1 / np.sqrt(var + eps), axis=0) + dvar * np.mean(-2 * (x - mean), axis=0)

  dx = dx_hat / np.sqrt(var + eps) + dvar * 2 * (x - mean) / N + dmean / N
```

```
  # ================================================================ #
  # 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.rand(*x.shape) < p) / p
    out = x * mask  # Apply mask

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

  elif mode == 'test':

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during test time.
    # ================================================================ #

    out = x # No dropout during test time

    # ================================================================ #
    # 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 = dout * mask  # Backpropagate through the 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 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
```

```python
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
  x_pad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode='constant')
  H_out = 1 + (H + 2 * pad - HH) // stride
  W_out = 1 + (W + 2 * pad - WW) // stride
  out = np.zeros((N, F, H_out, W_out))

  for n in range(N):
    for f in range(F):
      for h_out in range(H_out):
        for w_out in range(W_out):
          h_start = h_out * stride
          h_end = h_start + HH
          w_start = w_out * stride
          w_end = w_start + WW
          x_slice = x_pad[n, :, h_start:h_end, w_start:w_end]
          out[n, f, h_out, w_out] = np.sum(x_slice * 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.
  # ============================================================ #
  dxpad = np.zeros_like(xpad)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)
  for n in range(N):
    for f in range(F):
      db[f] += np.sum(dout[n, f])
      for h_out in range(out_height):
        for w_out in range(out_width):
          h_start = h_out * stride
          h_end = h_start + f_height
          w_start = w_out * stride
          w_end = w_start + f_width
          x_slice = xpad[n, :, h_start:h_end, w_start:w_end]
          dw[f] += x_slice * dout[n, f, h_out, w_out]
          dxpad[n, :, h_start:h_end, w_start:w_end] += w[f] * dout[n, f, h_out, w_out]
  dx = dxpad[:, :, pad:-pad, pad:-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_width, stride = pool_param['pool_height'], pool_param['pool_width'], 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 h_out in range(H_out):
        for w_out in range(W_out):
          h_start = h_out * stride
          h_end = h_start + pool_height
          w_start = w_out * stride
          w_end = w_start + pool_width
          x_slice = x[n, c, h_start:h_end, w_start:w_end]
          out[n, c, h_out, w_out] = np.max(x_slice)

  # ================================================================ #
  # 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']
  N, C, H, W = x.shape
  H_out = 1 + (H - pool_height) // stride
  W_out = 1 + (W - pool_width) // stride
  dx = np.zeros_like(x)

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the max pooling backward pass.
  # ================================================================ #
  for n in range(N):
    for c in range(C):
      for h_out in range(H_out):
        for w_out in range(W_out):
          h_start = h_out * stride
          h_end = h_start + pool_height
          w_start = w_out * stride
          w_end = w_start + pool_width
          x_slice = x[n, c, h_start:h_end, w_start:w_end]
          mask = (x_slice == np.max(x_slice))
          dx[n, c, h_start:h_end, w_start:w_end] = mask * dout[n, c, h_out, w_out]



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

  out, cache = None, None
  N, C, H, W = x.shape
  x_reshaped = x.transpose(0, 2, 3, 1).reshape(N * H * W, C)
  out_reshaped, cache = batchnorm_forward(x_reshaped, gamma, beta, bn_param)
  out = out_reshaped.reshape(N, H, W, C).transpose(0, 3, 1, 2)

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

  dx, dgamma, dbeta = None, None, None
  N, C, H, W = dout.shape
  dout_reshaped = dout.transpose(0, 2, 3, 1).reshape(N * H * W, C)
  dx_reshaped, dgamma, dbeta = batchnorm_backward(dout_reshaped, cache)
  dx = dx_reshaped.reshape(N, H, W, C).transpose(0, 3, 1, 2)


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

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

    return dx, dgamma, dbeta
```

```python
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 # Momentum update
  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.
  # ================================================================ #
  v_prev = v.copy()
  v = config['momentum'] * v - config['learning_rate'] * dw
  next_w = w - config['momentum'] * v_prev + (1 + config['momentum']) * v


  # ================================================================ #
  # 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.get('a', np.zeros_like(w))  # Get 'a' or initialize
  a = config['decay_rate'] * a + (1 - config['decay_rate']) * dw**2
  next_w = w - config['learning_rate'] * dw / (np.sqrt(a) + config['epsilon'])


  config['a'] = a

  # ================================================================ #
  # 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['t'] += 1
```

```
    # Compute moving averages of the gradient and its square
    config['v'] = config['beta1'] * config['v'] + (1 - config['beta1']) * dw
    config['a'] = config['beta2'] * config['a'] + (1 - config['beta2']) * (dw ** 2)

    # Bias correction
    v_corrected = config['v'] / (1 - config['beta1'] ** config['t'])
    a_corrected = config['a'] / (1 - config['beta2'] ** config['t'])

    # Adam update
    next_w = w - config['learning_rate'] * v_corrected / (np.sqrt(a_corrected) + config['epsilon'])


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

    return next_w, config
```

```python
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)
```

```
    pool_height, pool_width, stride = 2, 2, 2
    H_pool = 1 + (H - pool_height) // stride
    W_pool = 1 + (W - pool_width) // stride

    self.params['W2'] = weight_scale * np.random.randn(num_filters * H_pool * W_pool, 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".
    # ============================================================ #

    # conv - relu - pool
    a1, cache1 = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)

    # affine - relu
    a2, cache2 = affine_relu_forward(a1, W2, b2)

    # affine - softmax
    scores, cache3 = affine_forward(a2, 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)

    # affine - backward
    da2, dW3, db3 = affine_backward(dscores, cache3)

    # affine - relu - backward
    da1, dW2, db2 = affine_relu_backward(da2, cache2)

    # conv - relu - pool - backward
    dX, dW1, db1 = conv_relu_pool_backward(da1, cache1)

    # regularization
    loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2) + np.sum(W3**2))

    # add regularization gradients
    dW1 += self.reg * W1
    dW2 += self.reg * W2
    dW3 += self.reg * W3

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


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

    return loss, grads


pass
```

Spatial batch normalization

In fully connected networks, we performed batch normalization on the activations. To do something equivalent on CNNs, we modify batch normalization slightly.

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization accepts inputs of shape `(N, C, H, W)` and produces outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

How do we calculate the spatial averages? First, notice that for the `C` feature maps we have (i.e., the layer has `C` filters) that each of these ought to have its own batch norm statistics, since each feature map may be picking out very different features in the images. However, within a feature map, we may assume that across all inputs and across all locations in the feature map, there ought to be relatively similar first and second order statistics. Hence, one way to think of spatial batch-normalization is to reshape the `(N, C, H, W)` array as an `(N*H*W, C)` array and perform batch normalization on this array.

Since spatial batch norm and batch normalization are similar, it'd be good to at this point also copy and paste our prior implemented layers from HW #4. Please copy and paste your prior implemented code from HW #4 to start this assignment. If you did not correctly implement the layers in HW #4, you may collaborate with a classmate to use their implementations from HW #4. You may also visit TA or Prof OH to correct your implementation.

You'll want to copy and paste from HW #4:

```
- layers.py for your FC network layers, as well as batchnorm and dropout.
- layer_utils.py for your combined FC network layers.
- optim.py for your optimizers.
```

Be sure to place these in the `nndl/` directory so they're imported correctly. Note, as announced in class, we will not be releasing our solutions.

If you use your prior implementations of the batchnorm, then your spatial batchnorm implementation may be very short. Our implementations of the forward and backward pass are each 6 lines of code.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.conv_layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

Spatial batch normalization forward pass

Implement the forward pass, `spatial_batchnorm_forward` in `nndl/conv_layers.py`. Test your implementation by running the cell below.

In [2]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization
```

```
N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 9.8886138  10.10153535 10.28551476]
  Stds:  [4.34314621 3.56712323 3.96723925]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 3.85108612e-16 -9.40220124e-17 -1.27675648e-16]
  Stds:  [0.99999973 0.99999961 0.99999968]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.9999992  3.99999843 4.99999841]
```

Spatial batch normalization backward pass

Implement the backward pass, `spatial_batchnorm_backward` in `nndl/conv_layers.py`. Test your implementation by running the cell below.

In [3]:

```
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.6478858836245873e-08
dgamma error:  3.173515277047606e-11
dbeta error:  7.182571456288l2e-12
```

Convolutional neural network layers

In this notebook, we will build the convolutional neural network layers. This will be followed by a spatial batchnorm, and then in the final notebook of this assignment, we will train a CNN to further improve the validation accuracy on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [2]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.conv_layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Implementing CNN layers

Just as we implemented modular layers for fully connected networks, batch normalization, and dropout, we'll want to implement modular layers for convolutional neural networks. These layers are in `nndl/conv_layers.py`.

Convolutional forward pass

Begin by implementing a naive version of the forward pass of the CNN that uses `for` loops. This function is `conv_forward_naive` in `nndl/conv_layers.py`. Don't worry about efficiency of implementation. Later on, we provide a fast implementation of these layers. This version ought to test your understanding of convolution. In our implementation, there is a triple `for` loop.

After you implement `conv_forward_naive`, test your implementation by running the cell below.

In [3]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around 1e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

Convolutional backward pass

Now, implement a naive version of the backward pass of the CNN. The function is `conv_backward_naive` in `nndl/conv_layers.py`. Don't worry about efficiency of implementation. Later on, we provide a fast implementation of these layers. This version ought to test your understanding of convolution. In our implementation, there is a quadruple `for` loop.

After you implement `conv_backward_naive`, test your implementation by running the cell below.

In [4]:
```
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_forward_naive(x,w,b,conv_param)

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param)[0], b, dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around 1e-9'
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  6.756507376834252e-08
dw error:  9.493265573251991e-11
db error:  9.291237577180846e-12
```

Max pool forward pass

In this section, we will implement the forward pass of the max pool. The function is `max_pool_forward_naive` in `nndl/conv_layers.py`. Do not worry about the efficiency of implementation.

After you implement `max_pool_forward_naive`, test your implementation by running the cell below.

In [5]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be around 1e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

Max pool backward pass

In this section, you will implement the backward pass of the max pool. The function is `max_pool_backward_naive` in `nndl/conv_layers.py`. Do not worry about the efficiency of implementation.

After you implement `max_pool_backward_naive`, test your implementation by running the cell below.

In [6]:
```
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}
```

```
dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_param)[0], x, dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be around 1e-12
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.2756434562332255e-12
```

Fast implementation of the CNN layers

Implementing fast versions of the CNN layers can be difficult. We will provide you with the fast layers implemented by cs231n. They are provided in `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the cell below.

You should see pretty drastic speedups in the implementation of these layers. On our machine, the forward pass speeds up by 17x and the backward pass speeds up by 840x. Of course, these numbers will vary from machine to machine, as well as on your precise implementation of the naive layers.

In [7]:
```
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time

x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 1.882341s
Fast: 0.030535s
Speedup: 61.645398x
Difference:  2.2956330045957423e-11

Testing conv_backward_fast:
Naive: 2.692068s
Fast: 3.402141s
Speedup: 0.791286x
dx difference:  1.8017627433940875e-11
dw difference:  1.9502072246485964e-10
db difference:  0.0
```

In [8]:
```
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast

x = np.random.randn(100, 3, 32, 32)
```

```
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.138052s
fast: 0.002355s
speedup: 58.624279x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.324201s
speedup: 53.579652x
dx difference:  0.0
```

Implementation of cascaded layers

We've provided the following functions in `nndl/conv_layer_utils.py`:

- conv_relu_forward
- conv_relu_backward
- conv_relu_pool_forward
- conv_relu_pool_backward

These use the fast implementations of the conv net layers. You can test them below:

In [9]:
```
from nndl.conv_layer_utils import conv_relu_pool_forward, conv_relu_pool_backward

x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, conv_param, pool_param)[0], x,
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_param, pool_param)[0], w,
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_param, pool_param)[0], b,

print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu_pool
dx error:  2.4080216145717914e-08
dw error:  1.3979114766209628e-09
db error:  5.738176813089735e-10
```

In [10]:
```
from nndl.conv_layer_utils import conv_relu_forward, conv_relu_backward

x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)
```

```
dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_param)[0], b, dout)

print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu:
dx error:  1.1380980892303948e-08
dw error:  6.598675824619578e-10
db error:  1.0632105010823238e-11
```

What next?

We saw how helpful batch normalization was for training FC nets. In the next notebook, we'll implement a batch normalization for convolutional neural networks, and then finish off by implementing a CNN to improve our validation accuracy on CIFAR-10.

Convolutional neural networks

In this notebook, we'll put together our convolutional layers to implement a 3-layer CNN. Then, we'll ask you to implement a CNN that can achieve > 65% validation error on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

If you have not completed the Spatial BatchNorm Notebook, please see the following description from that notebook:

Please copy and paste your prior implemented code from HW #4 to start this assignment. If you did not correctly implement the layers in HW #4, you may collaborate with a classmate to use their layer implementations from HW #4. You may also visit TA or Prof OH to correct your implementation.

You'll want to copy and paste from HW #4:

```
- layers.py for your FC network layers, as well as batchnorm and dropout.
- layer_utils.py for your combined FC network layers.
- optim.py for your optimizers.
```

Be sure to place these in the `nndl/` directory so they're imported correctly. Note, as announced in class, we will not be releasing our solutions.

In [1]:
```
cd /Users/sujitsilas/Desktop/UCLA/Winter 2025/EE ENGR 247/Homeworks/HW5/
```

```
/Users/sujitsilas/Desktop/UCLA/Winter 2025/EE ENGR 247/Homeworks/HW5
```

In [2]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from nndl.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

In [15]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

Three layer CNN

In this notebook, you will implement a three layer CNN. The `ThreeLayerConvNet` class is in `nndl/cnn.py`. You'll need to modify that code for this section, including the initialization, as well as the calculation of the loss and gradients. You should be able to use the building blocks you have either earlier coded or that we have provided. Be sure to use the fast layers.

The architecture of this CNN will be:

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

We won't use batchnorm yet. You've also done enough of these to know how to debug; use the cells below.

Note: As we are implementing several layers CNN networks. The gradient error can be expected for the `eval_numerical_gradient()` function. If your `W1 max relative error` and `W2 max relative error` are around or below 0.01, they should be acceptable. Other errors should be less than 1e-5.

In [16]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, grads[param_name])))
```

```
W1 max relative error: 0.0003876121992847651
W2 max relative error: 0.006087720019066434
W3 max relative error: 0.00015610239462995478
b1 max relative error: 3.0482494482486916e-05
b2 max relative error: 6.380458561304692e-07
b3 max relative error: 1.3344205311457414e-09
```

Overfit small dataset

To check your CNN implementation, let's overfit a small dataset.

In [17]:
```
num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

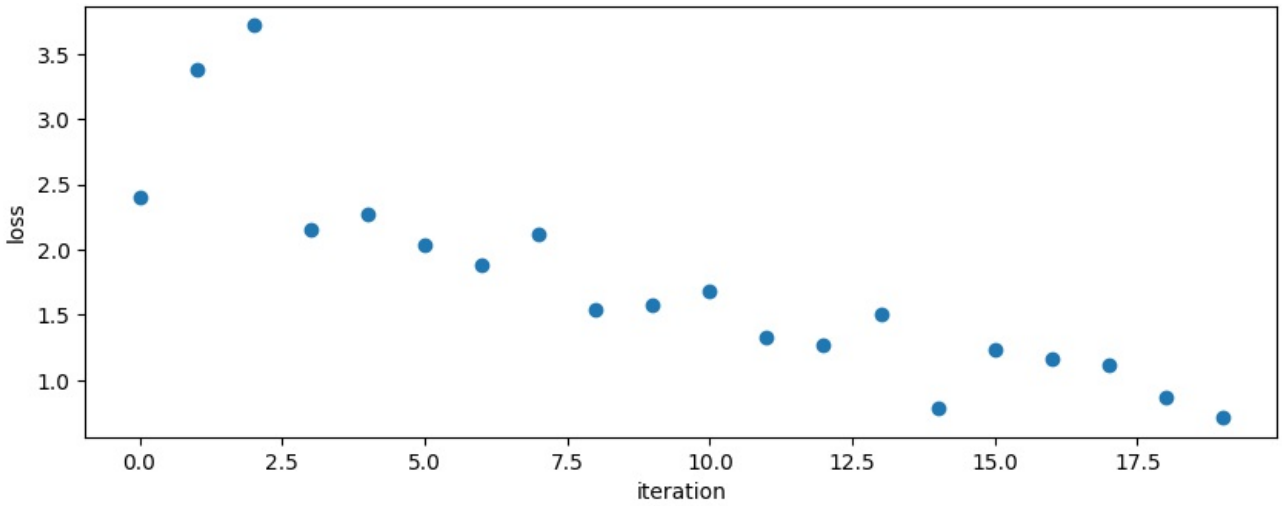
```
(Iteration 1 / 20) loss: 2.401445
(Epoch 0 / 10) train acc: 0.160000; val_acc: 0.112000
(Iteration 2 / 20) loss: 3.384418
(Epoch 1 / 10) train acc: 0.240000; val_acc: 0.133000
(Iteration 3 / 20) loss: 3.716092
(Iteration 4 / 20) loss: 2.159221
(Epoch 2 / 10) train acc: 0.350000; val_acc: 0.133000
(Iteration 5 / 20) loss: 2.267118
(Iteration 6 / 20) loss: 2.031910
(Epoch 3 / 10) train acc: 0.320000; val_acc: 0.138000
(Iteration 7 / 20) loss: 1.877286
(Iteration 8 / 20) loss: 2.123628
(Epoch 4 / 10) train acc: 0.450000; val_acc: 0.162000
(Iteration 9 / 20) loss: 1.535895
(Iteration 10 / 20) loss: 1.580020
(Epoch 5 / 10) train acc: 0.490000; val_acc: 0.142000
(Iteration 11 / 20) loss: 1.679795
(Iteration 12 / 20) loss: 1.332821
(Epoch 6 / 10) train acc: 0.590000; val_acc: 0.194000
(Iteration 13 / 20) loss: 1.275423
(Iteration 14 / 20) loss: 1.504342
(Epoch 7 / 10) train acc: 0.680000; val_acc: 0.218000
(Iteration 15 / 20) loss: 0.789805
(Iteration 16 / 20) loss: 1.229985
(Epoch 8 / 10) train acc: 0.670000; val_acc: 0.176000
(Iteration 17 / 20) loss: 1.163397
(Iteration 18 / 20) loss: 1.111179
(Epoch 9 / 10) train acc: 0.780000; val_acc: 0.214000
(Iteration 19 / 20) loss: 0.869874
(Iteration 20 / 20) loss: 0.715722
(Epoch 10 / 10) train acc: 0.800000; val_acc: 0.226000
```

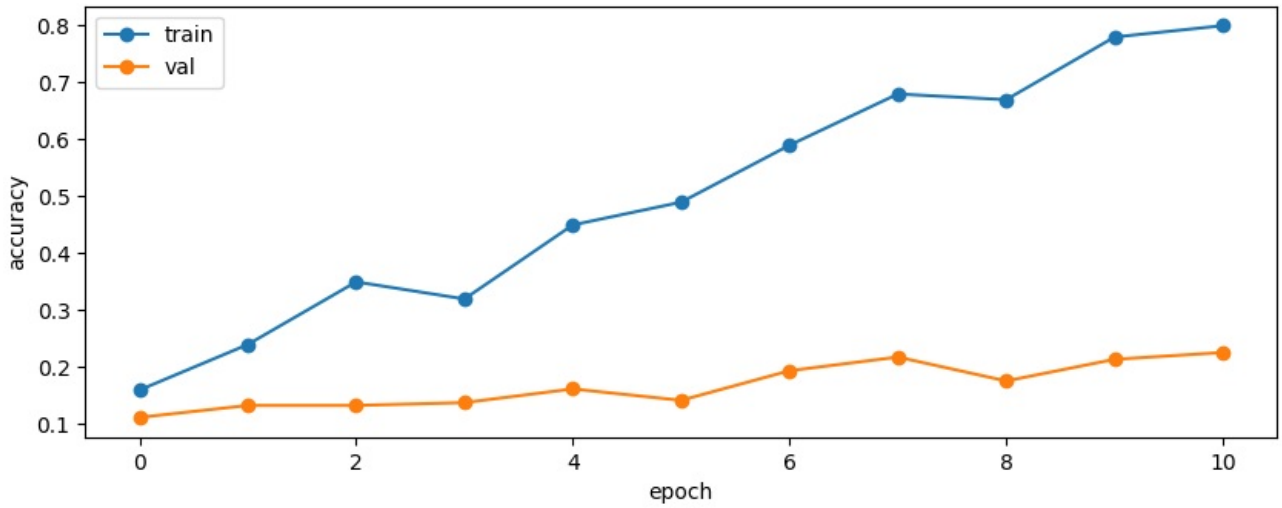
In [18]:

```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

```
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: savefi
g() got unexpected keyword argument "orientation" which is no longer supported as of 3.3 and will become an err
or two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: savefi
g() got unexpected keyword argument "dpi" which is no longer supported as of 3.3 and will become an error two m
inor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: savefi
g() got unexpected keyword argument "facecolor" which is no longer supported as of 3.3 and will become an error
two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: savefi
g() got unexpected keyword argument "edgecolor" which is no longer supported as of 3.3 and will become an error
two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: savefi
g() got unexpected keyword argument "bbox_inches_restore" which is no longer supported as of 3.3 and will becom
e an error two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
```





Train the network

Now we train the 3 layer CNN on CIFAR-10 and assess its accuracy.

In [19]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.304777
(Epoch 0 / 1) train acc: 0.085000; val_acc: 0.088000
(Iteration 21 / 980) loss: 2.343098
(Iteration 41 / 980) loss: 2.053573
(Iteration 61 / 980) loss: 2.170969
(Iteration 81 / 980) loss: 2.136866
(Iteration 101 / 980) loss: 1.953034
(Iteration 121 / 980) loss: 1.788268
(Iteration 141 / 980) loss: 1.951241
(Iteration 161 / 980) loss: 1.913286
(Iteration 181 / 980) loss: 1.729100
(Iteration 201 / 980) loss: 2.337749
(Iteration 221 / 980) loss: 1.633811
(Iteration 241 / 980) loss: 1.592094
(Iteration 261 / 980) loss: 1.806649
(Iteration 281 / 980) loss: 1.571482
(Iteration 301 / 980) loss: 1.605426
(Iteration 321 / 980) loss: 1.563820
(Iteration 341 / 980) loss: 1.562186
(Iteration 361 / 980) loss: 1.700869
(Iteration 381 / 980) loss: 1.618070
(Iteration 401 / 980) loss: 1.490276
(Iteration 421 / 980) loss: 1.642910
(Iteration 441 / 980) loss: 1.657358
(Iteration 461 / 980) loss: 1.583289
(Iteration 481 / 980) loss: 1.842153
(Iteration 501 / 980) loss: 1.744576
(Iteration 521 / 980) loss: 1.808263
(Iteration 541 / 980) loss: 1.774324
(Iteration 561 / 980) loss: 1.470594
(Iteration 581 / 980) loss: 1.354535
(Iteration 601 / 980) loss: 1.653459
(Iteration 621 / 980) loss: 1.496483
(Iteration 641 / 980) loss: 1.422213
(Iteration 661 / 980) loss: 1.665424
(Iteration 681 / 980) loss: 1.421259
(Iteration 701 / 980) loss: 1.654716
(Iteration 721 / 980) loss: 1.534159
(Iteration 741 / 980) loss: 1.509008
(Iteration 761 / 980) loss: 1.222114
(Iteration 781 / 980) loss: 1.609916
(Iteration 801 / 980) loss: 1.381218
(Iteration 821 / 980) loss: 1.603093
(Iteration 841 / 980) loss: 1.563135
(Iteration 861 / 980) loss: 1.488229
(Iteration 881 / 980) loss: 1.534100
(Iteration 901 / 980) loss: 1.640235
(Iteration 921 / 980) loss: 1.680015
(Iteration 941 / 980) loss: 1.544884
(Iteration 961 / 980) loss: 1.558549
(Epoch 1 / 1) train acc: 0.457000; val_acc: 0.479000
```

Get > 65% validation accuracy on CIFAR-10.

In the last part of the assignment, we'll now ask you to train a CNN to get better than 65% validation accuracy on CIFAR-10.

Things you should try:

- Filter size: Above we used 7x7; but VGGNet and onwards showed stacks of 3x3 filters are good.
- Number of filters: Above we used 32 filters. Do more or fewer do better?
- Batch normalization: Try adding spatial batch normalization after convolution layers and vanilla batch normalization aafter affine layers. Do your networks train faster?
- Network architecture: Can a deeper CNN do better? Consider these architectures:
  - [conv-relu-pool]xN - conv - relu - [affine]xM - [softmax or SVM]
  - [conv-relu-pool]XN - [affine]XM - [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN - [affine]xM - [softmax or SVM]

Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.

In [3]:
```
from cs231n.data_utils import load_CIFAR10

data = get_CIFAR10_data()
for k in data.keys():
    print('{}: {} '.format(k, data[k].shape))
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

In [4]:
```
def augment_data(X, y):
    """Augments CIFAR-10 data with horizontal flips and random crops."""
    X_aug = []
    y_aug = []

    for i in range(X.shape[0]):
        img = X[i].transpose(1, 2, 0)  # Convert to HWC format

        # Horizontal flip
        if np.random.rand() > 0.5:
            img_flipped = np.fliplr(img)
            X_aug.append(img_flipped.transpose(2, 0, 1))
            y_aug.append(y[i])

        # Random crop
        crop_size = 32
        x_offset = np.random.randint(0, 4)
        y_offset = np.random.randint(0, 4)

        img_padded = np.pad(img, ((4, 4), (4, 4), (0, 0)), mode='constant')
        img_cropped = img_padded[y_offset:y_offset + crop_size, x_offset:x_offset + crop_size, :]
        X_aug.append(img_cropped.transpose(2, 0, 1))
        y_aug.append(y[i])

    X_aug = np.concatenate([X, np.array(X_aug)], axis=0)
    y_aug = np.concatenate([y, np.array(y_aug)], axis=0)
    return X_aug, y_aug

# Augment training data
X_train_aug, y_train_aug = augment_data(data["X_train"], data["y_train"])

# Create augmented data dictionary
data_aug = {
    'X_train': X_train_aug,
    'y_train': y_train_aug,
    'X_val': data["X_val"],
    'y_val': data["y_val"],
    'X_test': data["X_test"],
    'y_test': data["y_test"]
}

# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #

# Define the model architecture
model = ThreeLayerConvNet(input_dim=(3, 32, 32), num_filters=64, filter_size=3,
                          hidden_dim=512, num_classes=10, weight_scale=0.001, reg=0.001, use_batchnorm=True)

# Define the solver parameters
solver = Solver(model, data_aug, #use augmented data
                num_epochs=20, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 0.001,
                },
                verbose=True, print_every=20)

# Train the model
solver.train()

# Print the validation accuracy
print('Best validation accuracy:', solver.best_val_acc)

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

```
(Iteration 1 / 24460) loss: 2.306876
(Epoch 0 / 20) train acc: 0.093000; val_acc: 0.119000
(Iteration 21 / 24460) loss: 1.968556
(Iteration 41 / 24460) loss: 1.992259
(Iteration 61 / 24460) loss: 1.876604
(Iteration 81 / 24460) loss: 1.597667
```

```
(Iteration 101 / 24460) loss: 1.623970
(Iteration 121 / 24460) loss: 1.846559
(Iteration 141 / 24460) loss: 1.563914
(Iteration 161 / 24460) loss: 1.742950
(Iteration 181 / 24460) loss: 1.841568
(Iteration 201 / 24460) loss: 1.713142
(Iteration 221 / 24460) loss: 1.522989
(Iteration 241 / 24460) loss: 1.362856
(Iteration 261 / 24460) loss: 1.529097
(Iteration 281 / 24460) loss: 1.761435
(Iteration 301 / 24460) loss: 1.737436
(Iteration 321 / 24460) loss: 1.818836
(Iteration 341 / 24460) loss: 1.487090
(Iteration 361 / 24460) loss: 1.521646
(Iteration 381 / 24460) loss: 1.424549
(Iteration 401 / 24460) loss: 1.651834
(Iteration 421 / 24460) loss: 1.385865
(Iteration 441 / 24460) loss: 1.480200
(Iteration 461 / 24460) loss: 1.645105
(Iteration 481 / 24460) loss: 1.579155
(Iteration 501 / 24460) loss: 1.520588
(Iteration 521 / 24460) loss: 1.647415
(Iteration 541 / 24460) loss: 1.642582
(Iteration 561 / 24460) loss: 1.405780
(Iteration 581 / 24460) loss: 1.372605
(Iteration 601 / 24460) loss: 1.690252
(Iteration 621 / 24460) loss: 1.316471
(Iteration 641 / 24460) loss: 1.303148
(Iteration 661 / 24460) loss: 1.605781
(Iteration 681 / 24460) loss: 1.452811
(Iteration 701 / 24460) loss: 1.363777
(Iteration 721 / 24460) loss: 1.608391
(Iteration 741 / 24460) loss: 1.441291
(Iteration 761 / 24460) loss: 1.141424
(Iteration 781 / 24460) loss: 1.375796
(Iteration 801 / 24460) loss: 1.325216
(Iteration 821 / 24460) loss: 1.460260
(Iteration 841 / 24460) loss: 1.398989
(Iteration 861 / 24460) loss: 1.490719
(Iteration 881 / 24460) loss: 1.457513
(Iteration 901 / 24460) loss: 1.390862
(Iteration 921 / 24460) loss: 1.437478
(Iteration 941 / 24460) loss: 1.466623
(Iteration 961 / 24460) loss: 1.361435
(Iteration 981 / 24460) loss: 1.292501
(Iteration 1001 / 24460) loss: 1.578588
(Iteration 1021 / 24460) loss: 1.355552
(Iteration 1041 / 24460) loss: 1.280713
(Iteration 1061 / 24460) loss: 1.468659
(Iteration 1081 / 24460) loss: 1.543792
(Iteration 1101 / 24460) loss: 1.382876
(Iteration 1121 / 24460) loss: 1.272412
(Iteration 1141 / 24460) loss: 1.097616
(Iteration 1161 / 24460) loss: 1.353857
(Iteration 1181 / 24460) loss: 1.581810
(Iteration 1201 / 24460) loss: 1.285922
(Iteration 1221 / 24460) loss: 1.488349
(Epoch 1 / 20) train acc: 0.561000; val_acc: 0.576000
(Iteration 1241 / 24460) loss: 1.347696
(Iteration 1261 / 24460) loss: 1.356189
(Iteration 1281 / 24460) loss: 1.454577
(Iteration 1301 / 24460) loss: 1.384634
(Iteration 1321 / 24460) loss: 1.067988
(Iteration 1341 / 24460) loss: 1.311321
(Iteration 1361 / 24460) loss: 1.319083
(Iteration 1381 / 24460) loss: 1.300423
(Iteration 1401 / 24460) loss: 1.204024
(Iteration 1421 / 24460) loss: 1.580231
(Iteration 1441 / 24460) loss: 1.349325
(Iteration 1461 / 24460) loss: 1.583886
(Iteration 1481 / 24460) loss: 1.513777
(Iteration 1501 / 24460) loss: 1.145278
(Iteration 1521 / 24460) loss: 1.128353
(Iteration 1541 / 24460) loss: 1.222516
(Iteration 1561 / 24460) loss: 1.456970
(Iteration 1581 / 24460) loss: 1.178496
(Iteration 1601 / 24460) loss: 1.266458
(Iteration 1621 / 24460) loss: 1.389638
(Iteration 1641 / 24460) loss: 1.363404
(Iteration 1661 / 24460) loss: 1.229295
(Iteration 1681 / 24460) loss: 1.256337
(Iteration 1701 / 24460) loss: 1.221898
(Iteration 1721 / 24460) loss: 1.338768
(Iteration 1741 / 24460) loss: 1.426326
(Iteration 1761 / 24460) loss: 1.540652
(Iteration 1781 / 24460) loss: 1.330854
(Iteration 1801 / 24460) loss: 1.185674
(Iteration 1821 / 24460) loss: 1.295520
(Iteration 1841 / 24460) loss: 1.036330
```

```
(Iteration 1861 / 24460) loss: 1.395794
(Iteration 1881 / 24460) loss: 1.137590
(Iteration 1901 / 24460) loss: 1.361948
(Iteration 1921 / 24460) loss: 1.181079
(Iteration 1941 / 24460) loss: 1.294370
(Iteration 1961 / 24460) loss: 1.530800
(Iteration 1981 / 24460) loss: 1.239763
(Iteration 2001 / 24460) loss: 1.253328
(Iteration 2021 / 24460) loss: 1.112606
(Iteration 2041 / 24460) loss: 1.173756
(Iteration 2061 / 24460) loss: 1.101437
(Iteration 2081 / 24460) loss: 1.080073
(Iteration 2101 / 24460) loss: 1.261534
(Iteration 2121 / 24460) loss: 1.081296
(Iteration 2141 / 24460) loss: 1.172701
(Iteration 2161 / 24460) loss: 1.054296
(Iteration 2181 / 24460) loss: 1.395740
(Iteration 2201 / 24460) loss: 1.146489
(Iteration 2221 / 24460) loss: 1.294128
(Iteration 2241 / 24460) loss: 1.176894
(Iteration 2261 / 24460) loss: 1.279096
(Iteration 2281 / 24460) loss: 1.151579
(Iteration 2301 / 24460) loss: 1.221663
(Iteration 2321 / 24460) loss: 1.276504
(Iteration 2341 / 24460) loss: 1.039846
(Iteration 2361 / 24460) loss: 1.185696
(Iteration 2381 / 24460) loss: 1.316065
(Iteration 2401 / 24460) loss: 1.084212
(Iteration 2421 / 24460) loss: 1.271406
(Iteration 2441 / 24460) loss: 1.249542
(Epoch 2 / 20) train acc: 0.624000; val_acc: 0.598000
(Iteration 2461 / 24460) loss: 1.056764
(Iteration 2481 / 24460) loss: 1.282084
(Iteration 2501 / 24460) loss: 1.431391
(Iteration 2521 / 24460) loss: 1.549483
(Iteration 2541 / 24460) loss: 1.354745
(Iteration 2561 / 24460) loss: 1.156140
(Iteration 2581 / 24460) loss: 1.264956
(Iteration 2601 / 24460) loss: 1.075385
(Iteration 2621 / 24460) loss: 1.095617
(Iteration 2641 / 24460) loss: 1.016367
(Iteration 2661 / 24460) loss: 1.391973
(Iteration 2681 / 24460) loss: 1.227830
(Iteration 2701 / 24460) loss: 1.097501
(Iteration 2721 / 24460) loss: 1.225333
(Iteration 2741 / 24460) loss: 1.290703
(Iteration 2761 / 24460) loss: 1.167535
(Iteration 2781 / 24460) loss: 1.198672
(Iteration 2801 / 24460) loss: 1.058708
(Iteration 2821 / 24460) loss: 1.189990
(Iteration 2841 / 24460) loss: 1.214469
(Iteration 2861 / 24460) loss: 1.043834
(Iteration 2881 / 24460) loss: 1.359238
(Iteration 2901 / 24460) loss: 1.088626
(Iteration 2921 / 24460) loss: 1.118719
(Iteration 2941 / 24460) loss: 1.278219
(Iteration 2961 / 24460) loss: 1.171085
(Iteration 2981 / 24460) loss: 1.377725
(Iteration 3001 / 24460) loss: 1.096921
(Iteration 3021 / 24460) loss: 1.144068
(Iteration 3041 / 24460) loss: 1.249327
(Iteration 3061 / 24460) loss: 1.220553
(Iteration 3081 / 24460) loss: 1.436901
(Iteration 3101 / 24460) loss: 1.316091
(Iteration 3121 / 24460) loss: 1.282944
(Iteration 3141 / 24460) loss: 1.029098
(Iteration 3161 / 24460) loss: 1.273283
(Iteration 3181 / 24460) loss: 1.393962
(Iteration 3201 / 24460) loss: 1.269004
(Iteration 3221 / 24460) loss: 0.989027
(Iteration 3241 / 24460) loss: 1.144613
(Iteration 3261 / 24460) loss: 1.158524
(Iteration 3281 / 24460) loss: 1.483371
(Iteration 3301 / 24460) loss: 1.160021
(Iteration 3321 / 24460) loss: 1.047326
(Iteration 3341 / 24460) loss: 1.227516
(Iteration 3361 / 24460) loss: 1.282071
(Iteration 3381 / 24460) loss: 1.194852
(Iteration 3401 / 24460) loss: 1.102651
(Iteration 3421 / 24460) loss: 1.039087
(Iteration 3441 / 24460) loss: 1.245268
(Iteration 3461 / 24460) loss: 1.094977
(Iteration 3481 / 24460) loss: 1.272504
(Iteration 3501 / 24460) loss: 1.137746
(Iteration 3521 / 24460) loss: 1.073925
(Iteration 3541 / 24460) loss: 1.274178
(Iteration 3561 / 24460) loss: 1.123111
(Iteration 3581 / 24460) loss: 1.016315
(Iteration 3601 / 24460) loss: 0.977294
```

```
(Iteration 3621 / 24460) loss: 1.109418
(Iteration 3641 / 24460) loss: 1.079412
(Iteration 3661 / 24460) loss: 1.306691
(Epoch 3 / 20) train acc: 0.615000; val_acc: 0.606000
(Iteration 3681 / 24460) loss: 1.186673
(Iteration 3701 / 24460) loss: 1.188733
(Iteration 3721 / 24460) loss: 0.961573
(Iteration 3741 / 24460) loss: 1.182203
(Iteration 3761 / 24460) loss: 1.123920
(Iteration 3781 / 24460) loss: 1.187859
(Iteration 3801 / 24460) loss: 1.249436
(Iteration 3821 / 24460) loss: 1.327765
(Iteration 3841 / 24460) loss: 1.069149
(Iteration 3861 / 24460) loss: 1.086173
(Iteration 3881 / 24460) loss: 1.120461
(Iteration 3901 / 24460) loss: 1.229361
(Iteration 3921 / 24460) loss: 0.971813
(Iteration 3941 / 24460) loss: 0.980358
(Iteration 3961 / 24460) loss: 1.176503
(Iteration 3981 / 24460) loss: 1.167497
(Iteration 4001 / 24460) loss: 1.065919
(Iteration 4021 / 24460) loss: 1.317747
(Iteration 4041 / 24460) loss: 1.386604
(Iteration 4061 / 24460) loss: 1.151038
(Iteration 4081 / 24460) loss: 1.364121
(Iteration 4101 / 24460) loss: 1.012557
(Iteration 4121 / 24460) loss: 1.030119
(Iteration 4141 / 24460) loss: 1.195229
(Iteration 4161 / 24460) loss: 1.269469
(Iteration 4181 / 24460) loss: 1.139683
(Iteration 4201 / 24460) loss: 1.122579
(Iteration 4221 / 24460) loss: 1.192275
(Iteration 4241 / 24460) loss: 1.252986
(Iteration 4261 / 24460) loss: 1.135524
(Iteration 4281 / 24460) loss: 1.283747
(Iteration 4301 / 24460) loss: 1.270721
(Iteration 4321 / 24460) loss: 1.186552
(Iteration 4341 / 24460) loss: 1.268468
(Iteration 4361 / 24460) loss: 1.076209
(Iteration 4381 / 24460) loss: 1.266939
(Iteration 4401 / 24460) loss: 1.132033
(Iteration 4421 / 24460) loss: 1.187137
(Iteration 4441 / 24460) loss: 1.124039
(Iteration 4461 / 24460) loss: 1.037668
(Iteration 4481 / 24460) loss: 0.946847
(Iteration 4501 / 24460) loss: 1.302835
(Iteration 4521 / 24460) loss: 1.358413
(Iteration 4541 / 24460) loss: 1.114341
(Iteration 4561 / 24460) loss: 1.211226
(Iteration 4581 / 24460) loss: 1.100552
(Iteration 4601 / 24460) loss: 1.133334
(Iteration 4621 / 24460) loss: 1.033083
(Iteration 4641 / 24460) loss: 0.888846
(Iteration 4661 / 24460) loss: 1.051547
(Iteration 4681 / 24460) loss: 1.117359
(Iteration 4701 / 24460) loss: 1.271176
(Iteration 4721 / 24460) loss: 1.424554
(Iteration 4741 / 24460) loss: 0.966229
(Iteration 4761 / 24460) loss: 1.128299
(Iteration 4781 / 24460) loss: 1.385073
(Iteration 4801 / 24460) loss: 1.141072
(Iteration 4821 / 24460) loss: 1.082224
(Iteration 4841 / 24460) loss: 1.258802
(Iteration 4861 / 24460) loss: 1.053847
(Iteration 4881 / 24460) loss: 1.013378
(Epoch 4 / 20) train acc: 0.673000; val_acc: 0.643000
(Iteration 4901 / 24460) loss: 1.133238
(Iteration 4921 / 24460) loss: 1.126092
(Iteration 4941 / 24460) loss: 1.502725
(Iteration 4961 / 24460) loss: 1.156968
(Iteration 4981 / 24460) loss: 1.049297
(Iteration 5001 / 24460) loss: 1.215352
(Iteration 5021 / 24460) loss: 0.973151
(Iteration 5041 / 24460) loss: 1.231094
(Iteration 5061 / 24460) loss: 1.261416
(Iteration 5081 / 24460) loss: 1.051167
(Iteration 5101 / 24460) loss: 1.251969
(Iteration 5121 / 24460) loss: 1.310049
(Iteration 5141 / 24460) loss: 1.117655
(Iteration 5161 / 24460) loss: 1.207392
(Iteration 5181 / 24460) loss: 0.998166
(Iteration 5201 / 24460) loss: 1.250700
(Iteration 5221 / 24460) loss: 1.230211
(Iteration 5241 / 24460) loss: 1.217338
(Iteration 5261 / 24460) loss: 1.062163
(Iteration 5281 / 24460) loss: 1.179421
(Iteration 5301 / 24460) loss: 1.392089
(Iteration 5321 / 24460) loss: 1.324026
(Iteration 5341 / 24460) loss: 0.920112
```

```
(Iteration 5361 / 24460) loss: 1.248448
(Iteration 5381 / 24460) loss: 1.112382
(Iteration 5401 / 24460) loss: 1.216238
(Iteration 5421 / 24460) loss: 0.994350
(Iteration 5441 / 24460) loss: 1.352212
(Iteration 5461 / 24460) loss: 1.140158
(Iteration 5481 / 24460) loss: 1.270309
(Iteration 5501 / 24460) loss: 1.065324
(Iteration 5521 / 24460) loss: 0.951950
(Iteration 5541 / 24460) loss: 1.315495
(Iteration 5561 / 24460) loss: 1.201802
(Iteration 5581 / 24460) loss: 0.995895
(Iteration 5601 / 24460) loss: 1.240260
(Iteration 5621 / 24460) loss: 1.317790
(Iteration 5641 / 24460) loss: 1.138205
(Iteration 5661 / 24460) loss: 1.268956
(Iteration 5681 / 24460) loss: 0.997571
(Iteration 5701 / 24460) loss: 1.025514
(Iteration 5721 / 24460) loss: 0.996115
(Iteration 5741 / 24460) loss: 1.142044
(Iteration 5761 / 24460) loss: 1.097030
(Iteration 5781 / 24460) loss: 0.920819
(Iteration 5801 / 24460) loss: 0.829411
(Iteration 5821 / 24460) loss: 1.098650
(Iteration 5841 / 24460) loss: 0.920301
(Iteration 5861 / 24460) loss: 1.099958
(Iteration 5881 / 24460) loss: 1.266457
(Iteration 5901 / 24460) loss: 0.894161
(Iteration 5921 / 24460) loss: 1.065915
(Iteration 5941 / 24460) loss: 1.143740
(Iteration 5961 / 24460) loss: 1.183432
(Iteration 5981 / 24460) loss: 0.963108
(Iteration 6001 / 24460) loss: 1.128657
(Iteration 6021 / 24460) loss: 1.501144
(Iteration 6041 / 24460) loss: 1.243106
(Iteration 6061 / 24460) loss: 1.118133
(Iteration 6081 / 24460) loss: 1.081439
(Iteration 6101 / 24460) loss: 1.099052
(Epoch 5 / 20) train acc: 0.663000; val_acc: 0.639000
(Iteration 6121 / 24460) loss: 1.129907
(Iteration 6141 / 24460) loss: 1.073058
(Iteration 6161 / 24460) loss: 1.141271
(Iteration 6181 / 24460) loss: 1.084283
(Iteration 6201 / 24460) loss: 1.194485
(Iteration 6221 / 24460) loss: 0.831768
(Iteration 6241 / 24460) loss: 1.106456
(Iteration 6261 / 24460) loss: 1.036564
(Iteration 6281 / 24460) loss: 1.079901
(Iteration 6301 / 24460) loss: 1.223262
(Iteration 6321 / 24460) loss: 1.032885
(Iteration 6341 / 24460) loss: 1.132415
(Iteration 6361 / 24460) loss: 1.098363
(Iteration 6381 / 24460) loss: 0.952874
(Iteration 6401 / 24460) loss: 0.998185
(Iteration 6421 / 24460) loss: 1.003338
(Iteration 6441 / 24460) loss: 1.086361
(Iteration 6461 / 24460) loss: 1.113846
(Iteration 6481 / 24460) loss: 1.139195
(Iteration 6501 / 24460) loss: 1.104181
(Iteration 6521 / 24460) loss: 1.158977
(Iteration 6541 / 24460) loss: 0.857722
(Iteration 6561 / 24460) loss: 1.051562
(Iteration 6581 / 24460) loss: 1.142748
(Iteration 6601 / 24460) loss: 0.953883
(Iteration 6621 / 24460) loss: 1.182391
(Iteration 6641 / 24460) loss: 1.129789
(Iteration 6661 / 24460) loss: 0.962286
(Iteration 6681 / 24460) loss: 1.138154
(Iteration 6701 / 24460) loss: 0.947399
(Iteration 6721 / 24460) loss: 0.938286
(Iteration 6741 / 24460) loss: 1.312306
(Iteration 6761 / 24460) loss: 1.092289
(Iteration 6781 / 24460) loss: 1.110781
(Iteration 6801 / 24460) loss: 1.212544
(Iteration 6821 / 24460) loss: 1.087380
(Iteration 6841 / 24460) loss: 1.088599
(Iteration 6861 / 24460) loss: 1.019982
(Iteration 6881 / 24460) loss: 1.158196
(Iteration 6901 / 24460) loss: 0.909006
(Iteration 6921 / 24460) loss: 1.091443
(Iteration 6941 / 24460) loss: 0.986882
(Iteration 6961 / 24460) loss: 1.210358
(Iteration 6981 / 24460) loss: 0.894493
(Iteration 7001 / 24460) loss: 1.524403
(Iteration 7021 / 24460) loss: 1.267861
(Iteration 7041 / 24460) loss: 1.174301
(Iteration 7061 / 24460) loss: 1.115855
(Iteration 7081 / 24460) loss: 1.108699
(Iteration 7101 / 24460) loss: 1.012441
```

```
(Iteration 7121 / 24460) loss: 1.084963
(Iteration 7141 / 24460) loss: 1.062684
(Iteration 7161 / 24460) loss: 1.274212
(Iteration 7181 / 24460) loss: 1.244564
(Iteration 7201 / 24460) loss: 1.095750
(Iteration 7221 / 24460) loss: 1.111947
(Iteration 7241 / 24460) loss: 1.189074
(Iteration 7261 / 24460) loss: 1.111293
(Iteration 7281 / 24460) loss: 1.276220
(Iteration 7301 / 24460) loss: 1.199976
(Iteration 7321 / 24460) loss: 1.154595
(Epoch 6 / 20) train acc: 0.679000; val_acc: 0.629000
(Iteration 7341 / 24460) loss: 1.028693
(Iteration 7361 / 24460) loss: 0.967818
(Iteration 7381 / 24460) loss: 1.205283
(Iteration 7401 / 24460) loss: 1.153289
(Iteration 7421 / 24460) loss: 1.313874
(Iteration 7441 / 24460) loss: 1.165045
(Iteration 7461 / 24460) loss: 1.017999
(Iteration 7481 / 24460) loss: 0.965494
(Iteration 7501 / 24460) loss: 0.801104
(Iteration 7521 / 24460) loss: 1.250936
(Iteration 7541 / 24460) loss: 0.897439
(Iteration 7561 / 24460) loss: 0.995711
(Iteration 7581 / 24460) loss: 0.979475
(Iteration 7601 / 24460) loss: 0.971818
(Iteration 7621 / 24460) loss: 1.004061
(Iteration 7641 / 24460) loss: 1.206641
(Iteration 7661 / 24460) loss: 1.064777
(Iteration 7681 / 24460) loss: 0.925063
(Iteration 7701 / 24460) loss: 1.063521
(Iteration 7721 / 24460) loss: 0.981187
(Iteration 7741 / 24460) loss: 1.115322
(Iteration 7761 / 24460) loss: 1.191560
(Iteration 7781 / 24460) loss: 1.044579
(Iteration 7801 / 24460) loss: 1.059569
(Iteration 7821 / 24460) loss: 0.856875
(Iteration 7841 / 24460) loss: 1.101357
(Iteration 7861 / 24460) loss: 1.299703
(Iteration 7881 / 24460) loss: 1.143184
(Iteration 7901 / 24460) loss: 0.951068
(Iteration 7921 / 24460) loss: 1.225451
(Iteration 7941 / 24460) loss: 0.890682
(Iteration 7961 / 24460) loss: 1.297413
(Iteration 7981 / 24460) loss: 1.204189
(Iteration 8001 / 24460) loss: 0.982908
(Iteration 8021 / 24460) loss: 1.088613
(Iteration 8041 / 24460) loss: 1.122282
(Iteration 8061 / 24460) loss: 0.943661
(Iteration 8081 / 24460) loss: 1.264588
(Iteration 8101 / 24460) loss: 1.069648
(Iteration 8121 / 24460) loss: 1.119463
(Iteration 8141 / 24460) loss: 1.193917
(Iteration 8161 / 24460) loss: 0.898159
(Iteration 8181 / 24460) loss: 0.866811
(Iteration 8201 / 24460) loss: 1.157122
(Iteration 8221 / 24460) loss: 0.967785
(Iteration 8241 / 24460) loss: 1.159309
(Iteration 8261 / 24460) loss: 1.302584
(Iteration 8281 / 24460) loss: 0.972762
(Iteration 8301 / 24460) loss: 1.085269
(Iteration 8321 / 24460) loss: 1.091957
(Iteration 8341 / 24460) loss: 1.091986
(Iteration 8361 / 24460) loss: 1.188257
(Iteration 8381 / 24460) loss: 1.071326
(Iteration 8401 / 24460) loss: 1.006811
(Iteration 8421 / 24460) loss: 1.190258
(Iteration 8441 / 24460) loss: 1.049709
(Iteration 8461 / 24460) loss: 0.788960
(Iteration 8481 / 24460) loss: 1.051102
(Iteration 8501 / 24460) loss: 0.977751
(Iteration 8521 / 24460) loss: 1.095597
(Iteration 8541 / 24460) loss: 0.973427
(Iteration 8561 / 24460) loss: 1.133476
(Epoch 7 / 20) train acc: 0.649000; val_acc: 0.642000
(Iteration 8581 / 24460) loss: 1.150982
(Iteration 8601 / 24460) loss: 1.258263
(Iteration 8621 / 24460) loss: 1.138344
(Iteration 8641 / 24460) loss: 1.047026
(Iteration 8661 / 24460) loss: 1.221310
(Iteration 8681 / 24460) loss: 1.067322
(Iteration 8701 / 24460) loss: 1.137405
(Iteration 8721 / 24460) loss: 1.238991
(Iteration 8741 / 24460) loss: 0.963916
(Iteration 8761 / 24460) loss: 1.303975
(Iteration 8781 / 24460) loss: 1.064782
(Iteration 8801 / 24460) loss: 0.818812
(Iteration 8821 / 24460) loss: 1.171133
(Iteration 8841 / 24460) loss: 0.881442
```

```
(Iteration 8861 / 24460) loss: 1.081328
(Iteration 8881 / 24460) loss: 1.021039
(Iteration 8901 / 24460) loss: 0.992392
(Iteration 8921 / 24460) loss: 1.159066
(Iteration 8941 / 24460) loss: 1.295421
(Iteration 8961 / 24460) loss: 1.187107
(Iteration 8981 / 24460) loss: 1.095315
(Iteration 9001 / 24460) loss: 1.087444
(Iteration 9021 / 24460) loss: 1.123139
(Iteration 9041 / 24460) loss: 0.999799
(Iteration 9061 / 24460) loss: 0.856076
(Iteration 9081 / 24460) loss: 0.980105
(Iteration 9101 / 24460) loss: 1.239787
(Iteration 9121 / 24460) loss: 1.180636
(Iteration 9141 / 24460) loss: 1.013291
(Iteration 9161 / 24460) loss: 1.057333
(Iteration 9181 / 24460) loss: 1.058848
(Iteration 9201 / 24460) loss: 0.961219
(Iteration 9221 / 24460) loss: 0.925803
(Iteration 9241 / 24460) loss: 1.293442
(Iteration 9261 / 24460) loss: 1.169147
(Iteration 9281 / 24460) loss: 1.154834
(Iteration 9301 / 24460) loss: 1.156065
(Iteration 9321 / 24460) loss: 0.962587
(Iteration 9341 / 24460) loss: 0.927212
(Iteration 9361 / 24460) loss: 0.960957
(Iteration 9381 / 24460) loss: 0.957741
(Iteration 9401 / 24460) loss: 0.938525
(Iteration 9421 / 24460) loss: 0.870595
(Iteration 9441 / 24460) loss: 0.991194
(Iteration 9461 / 24460) loss: 1.087964
(Iteration 9481 / 24460) loss: 1.121719
(Iteration 9501 / 24460) loss: 0.972605
(Iteration 9521 / 24460) loss: 1.045582
(Iteration 9541 / 24460) loss: 1.098090
(Iteration 9561 / 24460) loss: 1.069079
(Iteration 9581 / 24460) loss: 0.969363
(Iteration 9601 / 24460) loss: 1.048867
(Iteration 9621 / 24460) loss: 0.964241
(Iteration 9641 / 24460) loss: 1.177604
(Iteration 9661 / 24460) loss: 1.166654
(Iteration 9681 / 24460) loss: 0.936888
(Iteration 9701 / 24460) loss: 1.103031
(Iteration 9721 / 24460) loss: 1.032414
(Iteration 9741 / 24460) loss: 1.248121
(Iteration 9761 / 24460) loss: 1.333015
(Iteration 9781 / 24460) loss: 0.912411
(Epoch 8 / 20) train acc: 0.687000; val_acc: 0.638000
(Iteration 9801 / 24460) loss: 1.097412
(Iteration 9821 / 24460) loss: 0.985956
(Iteration 9841 / 24460) loss: 1.028175
(Iteration 9861 / 24460) loss: 1.104003
(Iteration 9881 / 24460) loss: 1.031309
(Iteration 9901 / 24460) loss: 0.939893
(Iteration 9921 / 24460) loss: 1.059498
(Iteration 9941 / 24460) loss: 1.276190
(Iteration 9961 / 24460) loss: 1.183860
(Iteration 9981 / 24460) loss: 1.045031
(Iteration 10001 / 24460) loss: 1.038027
(Iteration 10021 / 24460) loss: 1.024074
(Iteration 10041 / 24460) loss: 1.023974
(Iteration 10061 / 24460) loss: 1.171422
(Iteration 10081 / 24460) loss: 1.057586
(Iteration 10101 / 24460) loss: 1.102446
(Iteration 10121 / 24460) loss: 1.198798
(Iteration 10141 / 24460) loss: 1.129857
(Iteration 10161 / 24460) loss: 1.174973
(Iteration 10181 / 24460) loss: 1.038794
(Iteration 10201 / 24460) loss: 0.967806
(Iteration 10221 / 24460) loss: 1.148438
(Iteration 10241 / 24460) loss: 1.113279
(Iteration 10261 / 24460) loss: 1.135742
(Iteration 10281 / 24460) loss: 1.024751
(Iteration 10301 / 24460) loss: 0.998011
(Iteration 10321 / 24460) loss: 1.018965
(Iteration 10341 / 24460) loss: 0.968576
(Iteration 10361 / 24460) loss: 1.078993
(Iteration 10381 / 24460) loss: 0.996548
(Iteration 10401 / 24460) loss: 1.157222
(Iteration 10421 / 24460) loss: 1.067188
(Iteration 10441 / 24460) loss: 1.020597
(Iteration 10461 / 24460) loss: 1.251286
(Iteration 10481 / 24460) loss: 1.223498
(Iteration 10501 / 24460) loss: 1.138038
(Iteration 10521 / 24460) loss: 1.183617
(Iteration 10541 / 24460) loss: 0.958307
(Iteration 10561 / 24460) loss: 1.253840
(Iteration 10581 / 24460) loss: 0.989114
(Iteration 10601 / 24460) loss: 1.278203
```

```
(Iteration 10621 / 24460) loss: 1.031079
(Iteration 10641 / 24460) loss: 1.096198
(Iteration 10661 / 24460) loss: 1.052968
(Iteration 10681 / 24460) loss: 1.117115
(Iteration 10701 / 24460) loss: 1.200436
(Iteration 10721 / 24460) loss: 1.249390
(Iteration 10741 / 24460) loss: 1.130456
(Iteration 10761 / 24460) loss: 0.944164
(Iteration 10781 / 24460) loss: 1.056075
(Iteration 10801 / 24460) loss: 1.099870
(Iteration 10821 / 24460) loss: 1.233391
(Iteration 10841 / 24460) loss: 1.011289
(Iteration 10861 / 24460) loss: 0.892174
(Iteration 10881 / 24460) loss: 0.870685
(Iteration 10901 / 24460) loss: 1.046534
(Iteration 10921 / 24460) loss: 1.216591
(Iteration 10941 / 24460) loss: 0.972797
(Iteration 10961 / 24460) loss: 0.991299
(Iteration 10981 / 24460) loss: 1.052651
(Iteration 11001 / 24460) loss: 0.981184
(Epoch 9 / 20) train acc: 0.678000; val_acc: 0.636000
(Iteration 11021 / 24460) loss: 1.066156
(Iteration 11041 / 24460) loss: 1.188949
(Iteration 11061 / 24460) loss: 1.132900
(Iteration 11081 / 24460) loss: 1.087312
(Iteration 11101 / 24460) loss: 1.003873
(Iteration 11121 / 24460) loss: 1.026794
(Iteration 11141 / 24460) loss: 0.964611
(Iteration 11161 / 24460) loss: 1.047817
(Iteration 11181 / 24460) loss: 1.017285
(Iteration 11201 / 24460) loss: 0.943714
(Iteration 11221 / 24460) loss: 1.036221
(Iteration 11241 / 24460) loss: 0.974081
(Iteration 11261 / 24460) loss: 0.804067
(Iteration 11281 / 24460) loss: 0.898356
(Iteration 11301 / 24460) loss: 0.977749
(Iteration 11321 / 24460) loss: 1.059252
(Iteration 11341 / 24460) loss: 0.953022
(Iteration 11361 / 24460) loss: 1.086798
(Iteration 11381 / 24460) loss: 1.104006
(Iteration 11401 / 24460) loss: 1.050733
(Iteration 11421 / 24460) loss: 1.119143
(Iteration 11441 / 24460) loss: 0.907106
(Iteration 11461 / 24460) loss: 1.200150
(Iteration 11481 / 24460) loss: 1.211437
(Iteration 11501 / 24460) loss: 0.754366
(Iteration 11521 / 24460) loss: 0.991990
(Iteration 11541 / 24460) loss: 0.978120
(Iteration 11561 / 24460) loss: 0.968699
(Iteration 11581 / 24460) loss: 1.051384
(Iteration 11601 / 24460) loss: 1.001912
(Iteration 11621 / 24460) loss: 1.070563
(Iteration 11641 / 24460) loss: 0.896923
(Iteration 11661 / 24460) loss: 0.887776
(Iteration 11681 / 24460) loss: 1.282454
(Iteration 11701 / 24460) loss: 1.294549
(Iteration 11721 / 24460) loss: 1.202442
(Iteration 11741 / 24460) loss: 1.036215
(Iteration 11761 / 24460) loss: 1.004647
(Iteration 11781 / 24460) loss: 1.047714
(Iteration 11801 / 24460) loss: 0.826549
(Iteration 11821 / 24460) loss: 0.992534
(Iteration 11841 / 24460) loss: 1.088050
(Iteration 11861 / 24460) loss: 1.044917
(Iteration 11881 / 24460) loss: 0.987356
(Iteration 11901 / 24460) loss: 0.976187
(Iteration 11921 / 24460) loss: 1.224390
(Iteration 11941 / 24460) loss: 1.115215
(Iteration 11961 / 24460) loss: 1.037788
(Iteration 11981 / 24460) loss: 1.080773
(Iteration 12001 / 24460) loss: 1.164847
(Iteration 12021 / 24460) loss: 0.881558
(Iteration 12041 / 24460) loss: 0.903563
(Iteration 12061 / 24460) loss: 1.149447
(Iteration 12081 / 24460) loss: 1.112572
(Iteration 12101 / 24460) loss: 0.765414
(Iteration 12121 / 24460) loss: 1.056328
(Iteration 12141 / 24460) loss: 1.036165
(Iteration 12161 / 24460) loss: 0.964383
(Iteration 12181 / 24460) loss: 1.130474
(Iteration 12201 / 24460) loss: 0.966891
(Iteration 12221 / 24460) loss: 0.860122
(Epoch 10 / 20) train acc: 0.713000; val_acc: 0.656000
(Iteration 12241 / 24460) loss: 1.052770
(Iteration 12261 / 24460) loss: 0.970801
(Iteration 12281 / 24460) loss: 1.192567
(Iteration 12301 / 24460) loss: 1.134865
(Iteration 12321 / 24460) loss: 1.102948
(Iteration 12341 / 24460) loss: 0.923098
```

```
(Iteration 12361 / 24460) loss: 1.197572
(Iteration 12381 / 24460) loss: 1.135824
(Iteration 12401 / 24460) loss: 1.090383
(Iteration 12421 / 24460) loss: 1.107946
(Iteration 12441 / 24460) loss: 0.999766
(Iteration 12461 / 24460) loss: 1.171789
(Iteration 12481 / 24460) loss: 0.831641
(Iteration 12501 / 24460) loss: 1.035091
(Iteration 12521 / 24460) loss: 1.118523
(Iteration 12541 / 24460) loss: 1.117638
(Iteration 12561 / 24460) loss: 1.085577
(Iteration 12581 / 24460) loss: 1.377333
(Iteration 12601 / 24460) loss: 1.230613
(Iteration 12621 / 24460) loss: 1.076819
(Iteration 12641 / 24460) loss: 1.154344
(Iteration 12661 / 24460) loss: 1.133345
(Iteration 12681 / 24460) loss: 1.100039
(Iteration 12701 / 24460) loss: 1.063743
(Iteration 12721 / 24460) loss: 1.059160
(Iteration 12741 / 24460) loss: 1.087043
(Iteration 12761 / 24460) loss: 1.188213
(Iteration 12781 / 24460) loss: 0.947083
(Iteration 12801 / 24460) loss: 0.874229
(Iteration 12821 / 24460) loss: 1.014170
(Iteration 12841 / 24460) loss: 1.121244
(Iteration 12861 / 24460) loss: 1.063940
(Iteration 12881 / 24460) loss: 1.253822
(Iteration 12901 / 24460) loss: 1.055508
(Iteration 12921 / 24460) loss: 0.934860
(Iteration 12941 / 24460) loss: 1.144986
(Iteration 12961 / 24460) loss: 1.000696
(Iteration 12981 / 24460) loss: 0.997892
(Iteration 13001 / 24460) loss: 0.841473
(Iteration 13021 / 24460) loss: 1.095416
(Iteration 13041 / 24460) loss: 0.959957
(Iteration 13061 / 24460) loss: 0.961732
(Iteration 13081 / 24460) loss: 0.921939
(Iteration 13101 / 24460) loss: 0.987105
(Iteration 13121 / 24460) loss: 0.861549
(Iteration 13141 / 24460) loss: 1.016953
(Iteration 13161 / 24460) loss: 0.951785
(Iteration 13181 / 24460) loss: 1.215004
(Iteration 13201 / 24460) loss: 0.849733
(Iteration 13221 / 24460) loss: 0.910898
(Iteration 13241 / 24460) loss: 0.997898
(Iteration 13261 / 24460) loss: 0.916990
(Iteration 13281 / 24460) loss: 0.996246
(Iteration 13301 / 24460) loss: 1.134790
(Iteration 13321 / 24460) loss: 0.993735
(Iteration 13341 / 24460) loss: 1.095465
(Iteration 13361 / 24460) loss: 1.073289
(Iteration 13381 / 24460) loss: 1.264105
(Iteration 13401 / 24460) loss: 0.839522
(Iteration 13421 / 24460) loss: 1.034215
(Iteration 13441 / 24460) loss: 0.916255
(Epoch 11 / 20) train acc: 0.675000; val_acc: 0.653000
(Iteration 13461 / 24460) loss: 0.871069
(Iteration 13481 / 24460) loss: 1.073814
(Iteration 13501 / 24460) loss: 1.057457
(Iteration 13521 / 24460) loss: 1.045322
(Iteration 13541 / 24460) loss: 1.129634
(Iteration 13561 / 24460) loss: 0.868889
(Iteration 13581 / 24460) loss: 1.064611
(Iteration 13601 / 24460) loss: 1.016197
(Iteration 13621 / 24460) loss: 1.267896
(Iteration 13641 / 24460) loss: 0.863097
(Iteration 13661 / 24460) loss: 0.966914
(Iteration 13681 / 24460) loss: 1.016997
(Iteration 13701 / 24460) loss: 1.153994
(Iteration 13721 / 24460) loss: 0.999021
(Iteration 13741 / 24460) loss: 1.083256
(Iteration 13761 / 24460) loss: 0.951712
(Iteration 13781 / 24460) loss: 1.209516
(Iteration 13801 / 24460) loss: 1.083192
(Iteration 13821 / 24460) loss: 1.016923
(Iteration 13841 / 24460) loss: 0.916959
(Iteration 13861 / 24460) loss: 1.065910
(Iteration 13881 / 24460) loss: 1.206309
(Iteration 13901 / 24460) loss: 1.080893
(Iteration 13921 / 24460) loss: 1.043711
(Iteration 13941 / 24460) loss: 0.982596
(Iteration 13961 / 24460) loss: 0.990555
(Iteration 13981 / 24460) loss: 1.051925
(Iteration 14001 / 24460) loss: 1.027502
(Iteration 14021 / 24460) loss: 1.207919
(Iteration 14041 / 24460) loss: 1.084263
(Iteration 14061 / 24460) loss: 1.169087
(Iteration 14081 / 24460) loss: 0.903914
(Iteration 14101 / 24460) loss: 0.914025
```

```
(Iteration 14121 / 24460) loss: 0.998631
(Iteration 14141 / 24460) loss: 1.200306
(Iteration 14161 / 24460) loss: 1.104157
(Iteration 14181 / 24460) loss: 0.924816
(Iteration 14201 / 24460) loss: 1.292522
(Iteration 14221 / 24460) loss: 1.028224
(Iteration 14241 / 24460) loss: 0.968215
(Iteration 14261 / 24460) loss: 1.000287
(Iteration 14281 / 24460) loss: 1.141932
(Iteration 14301 / 24460) loss: 1.084227
(Iteration 14321 / 24460) loss: 0.952204
(Iteration 14341 / 24460) loss: 1.030548
(Iteration 14361 / 24460) loss: 1.031550
(Iteration 14381 / 24460) loss: 0.996652
(Iteration 14401 / 24460) loss: 1.282737
(Iteration 14421 / 24460) loss: 0.996048
(Iteration 14441 / 24460) loss: 1.044553
(Iteration 14461 / 24460) loss: 0.840014
(Iteration 14481 / 24460) loss: 1.122835
(Iteration 14501 / 24460) loss: 1.082399
(Iteration 14521 / 24460) loss: 1.061275
(Iteration 14541 / 24460) loss: 0.966500
(Iteration 14561 / 24460) loss: 1.036658
(Iteration 14581 / 24460) loss: 1.038086
(Iteration 14601 / 24460) loss: 1.014901
(Iteration 14621 / 24460) loss: 1.002290
(Iteration 14641 / 24460) loss: 0.901779
(Iteration 14661 / 24460) loss: 1.005527
(Epoch 12 / 20) train acc: 0.678000; val_acc: 0.663000
(Iteration 14681 / 24460) loss: 0.881486
(Iteration 14701 / 24460) loss: 1.287045
(Iteration 14721 / 24460) loss: 0.923514
(Iteration 14741 / 24460) loss: 1.108627
(Iteration 14761 / 24460) loss: 0.968628
(Iteration 14781 / 24460) loss: 1.192262
(Iteration 14801 / 24460) loss: 1.116888
(Iteration 14821 / 24460) loss: 1.073804
(Iteration 14841 / 24460) loss: 1.087936
(Iteration 14861 / 24460) loss: 1.025539
(Iteration 14881 / 24460) loss: 0.926272
(Iteration 14901 / 24460) loss: 1.123477
(Iteration 14921 / 24460) loss: 1.114028
(Iteration 14941 / 24460) loss: 0.929525
(Iteration 14961 / 24460) loss: 0.875629
(Iteration 14981 / 24460) loss: 0.892974
(Iteration 15001 / 24460) loss: 0.931920
(Iteration 15021 / 24460) loss: 1.230533
(Iteration 15041 / 24460) loss: 0.861700
(Iteration 15061 / 24460) loss: 1.206768
(Iteration 15081 / 24460) loss: 1.024184
(Iteration 15101 / 24460) loss: 1.204219
(Iteration 15121 / 24460) loss: 1.081250
(Iteration 15141 / 24460) loss: 1.242850
(Iteration 15161 / 24460) loss: 0.971097
(Iteration 15181 / 24460) loss: 1.046391
(Iteration 15201 / 24460) loss: 0.888377
(Iteration 15221 / 24460) loss: 1.098030
(Iteration 15241 / 24460) loss: 1.132702
(Iteration 15261 / 24460) loss: 1.199652
(Iteration 15281 / 24460) loss: 1.372494
(Iteration 15301 / 24460) loss: 1.136523
(Iteration 15321 / 24460) loss: 1.027049
(Iteration 15341 / 24460) loss: 1.031172
(Iteration 15361 / 24460) loss: 1.032550
(Iteration 15381 / 24460) loss: 0.963458
(Iteration 15401 / 24460) loss: 0.905759
(Iteration 15421 / 24460) loss: 1.043518
(Iteration 15441 / 24460) loss: 1.114398
(Iteration 15461 / 24460) loss: 1.114779
(Iteration 15481 / 24460) loss: 0.912827
(Iteration 15501 / 24460) loss: 1.059740
(Iteration 15521 / 24460) loss: 0.867980
(Iteration 15541 / 24460) loss: 1.112523
(Iteration 15561 / 24460) loss: 1.060135
(Iteration 15581 / 24460) loss: 1.059116
(Iteration 15601 / 24460) loss: 1.129436
(Iteration 15621 / 24460) loss: 0.994034
(Iteration 15641 / 24460) loss: 1.120166
(Iteration 15661 / 24460) loss: 1.228634
(Iteration 15681 / 24460) loss: 0.998708
(Iteration 15701 / 24460) loss: 0.917941
(Iteration 15721 / 24460) loss: 0.964858
(Iteration 15741 / 24460) loss: 1.027250
(Iteration 15761 / 24460) loss: 1.111651
(Iteration 15781 / 24460) loss: 0.807349
(Iteration 15801 / 24460) loss: 0.939413
(Iteration 15821 / 24460) loss: 1.083140
(Iteration 15841 / 24460) loss: 1.024578
(Iteration 15861 / 24460) loss: 1.069784
```

```
(Iteration 15881 / 24460) loss: 1.195064
(Epoch 13 / 20) train acc: 0.695000; val_acc: 0.660000
(Iteration 15901 / 24460) loss: 0.916705
(Iteration 15921 / 24460) loss: 0.915117
(Iteration 15941 / 24460) loss: 1.171014
(Iteration 15961 / 24460) loss: 1.074659
(Iteration 15981 / 24460) loss: 1.004918
(Iteration 16001 / 24460) loss: 0.950459
(Iteration 16021 / 24460) loss: 0.943545
(Iteration 16041 / 24460) loss: 0.911328
(Iteration 16061 / 24460) loss: 1.155470
(Iteration 16081 / 24460) loss: 1.055234
(Iteration 16101 / 24460) loss: 0.878317
(Iteration 16121 / 24460) loss: 0.959692
(Iteration 16141 / 24460) loss: 1.121924
(Iteration 16161 / 24460) loss: 0.928708
(Iteration 16181 / 24460) loss: 1.149211
(Iteration 16201 / 24460) loss: 1.051510
(Iteration 16221 / 24460) loss: 0.997013
(Iteration 16241 / 24460) loss: 1.009884
(Iteration 16261 / 24460) loss: 0.836328
(Iteration 16281 / 24460) loss: 0.925528
(Iteration 16301 / 24460) loss: 1.017415
(Iteration 16321 / 24460) loss: 1.120647
(Iteration 16341 / 24460) loss: 1.213527
(Iteration 16361 / 24460) loss: 1.047769
(Iteration 16381 / 24460) loss: 0.788155
(Iteration 16401 / 24460) loss: 1.067019
(Iteration 16421 / 24460) loss: 1.094355
(Iteration 16441 / 24460) loss: 0.910270
(Iteration 16461 / 24460) loss: 1.004971
(Iteration 16481 / 24460) loss: 0.889767
(Iteration 16501 / 24460) loss: 0.986653
(Iteration 16521 / 24460) loss: 1.027288
(Iteration 16541 / 24460) loss: 0.673533
(Iteration 16561 / 24460) loss: 0.949765
(Iteration 16581 / 24460) loss: 1.099160
(Iteration 16601 / 24460) loss: 1.065408
(Iteration 16621 / 24460) loss: 1.278286
(Iteration 16641 / 24460) loss: 1.040013
(Iteration 16661 / 24460) loss: 0.887716
(Iteration 16681 / 24460) loss: 0.836183
(Iteration 16701 / 24460) loss: 1.001875
(Iteration 16721 / 24460) loss: 1.113995
(Iteration 16741 / 24460) loss: 0.858828
(Iteration 16761 / 24460) loss: 0.966579
(Iteration 16781 / 24460) loss: 0.895353
(Iteration 16801 / 24460) loss: 0.954584
(Iteration 16821 / 24460) loss: 1.030165
(Iteration 16841 / 24460) loss: 0.919301
(Iteration 16861 / 24460) loss: 0.854855
(Iteration 16881 / 24460) loss: 0.869481
(Iteration 16901 / 24460) loss: 1.120940
(Iteration 16921 / 24460) loss: 1.011558
(Iteration 16941 / 24460) loss: 0.911969
(Iteration 16961 / 24460) loss: 1.012530
(Iteration 16981 / 24460) loss: 0.890771
(Iteration 17001 / 24460) loss: 1.054197
(Iteration 17021 / 24460) loss: 0.801266
(Iteration 17041 / 24460) loss: 0.803925
(Iteration 17061 / 24460) loss: 1.128880
(Iteration 17081 / 24460) loss: 0.808644
(Iteration 17101 / 24460) loss: 1.176471
(Iteration 17121 / 24460) loss: 0.939181
(Epoch 14 / 20) train acc: 0.685000; val_acc: 0.654000
(Iteration 17141 / 24460) loss: 0.994614
(Iteration 17161 / 24460) loss: 0.882954
(Iteration 17181 / 24460) loss: 0.851716
(Iteration 17201 / 24460) loss: 1.254913
(Iteration 17221 / 24460) loss: 0.937979
(Iteration 17241 / 24460) loss: 0.895342
(Iteration 17261 / 24460) loss: 1.032736
(Iteration 17281 / 24460) loss: 1.086330
(Iteration 17301 / 24460) loss: 1.171603
(Iteration 17321 / 24460) loss: 0.798791
(Iteration 17341 / 24460) loss: 1.064891
(Iteration 17361 / 24460) loss: 0.967917
(Iteration 17381 / 24460) loss: 1.028399
(Iteration 17401 / 24460) loss: 1.111676
(Iteration 17421 / 24460) loss: 1.041382
(Iteration 17441 / 24460) loss: 0.941041
(Iteration 17461 / 24460) loss: 0.831273
(Iteration 17481 / 24460) loss: 0.979104
(Iteration 17501 / 24460) loss: 0.798523
(Iteration 17521 / 24460) loss: 0.968189
(Iteration 17541 / 24460) loss: 0.855786
(Iteration 17561 / 24460) loss: 0.974229
(Iteration 17581 / 24460) loss: 0.936157
(Iteration 17601 / 24460) loss: 1.053959
```

```
(Iteration 17621 / 24460) loss: 1.102445
(Iteration 17641 / 24460) loss: 1.121313
(Iteration 17661 / 24460) loss: 0.924026
(Iteration 17681 / 24460) loss: 1.177346
(Iteration 17701 / 24460) loss: 0.884149
(Iteration 17721 / 24460) loss: 1.071664
(Iteration 17741 / 24460) loss: 0.970736
(Iteration 17761 / 24460) loss: 1.042843
(Iteration 17781 / 24460) loss: 1.125770
(Iteration 17801 / 24460) loss: 1.247933
(Iteration 17821 / 24460) loss: 0.830458
(Iteration 17841 / 24460) loss: 1.056150
(Iteration 17861 / 24460) loss: 0.971811
(Iteration 17881 / 24460) loss: 0.825947
(Iteration 17901 / 24460) loss: 1.136552
(Iteration 17921 / 24460) loss: 0.953373
(Iteration 17941 / 24460) loss: 0.901938
(Iteration 17961 / 24460) loss: 0.916282
(Iteration 17981 / 24460) loss: 0.956023
(Iteration 18001 / 24460) loss: 0.859028
(Iteration 18021 / 24460) loss: 1.110694
(Iteration 18041 / 24460) loss: 1.050298
(Iteration 18061 / 24460) loss: 0.941958
(Iteration 18081 / 24460) loss: 1.004586
(Iteration 18101 / 24460) loss: 1.210491
(Iteration 18121 / 24460) loss: 0.869880
(Iteration 18141 / 24460) loss: 1.017760
(Iteration 18161 / 24460) loss: 0.975835
(Iteration 18181 / 24460) loss: 0.850423
(Iteration 18201 / 24460) loss: 0.981225
(Iteration 18221 / 24460) loss: 0.849840
(Iteration 18241 / 24460) loss: 0.989565
(Iteration 18261 / 24460) loss: 1.059574
(Iteration 18281 / 24460) loss: 1.090118
(Iteration 18301 / 24460) loss: 0.848001
(Iteration 18321 / 24460) loss: 0.977244
(Iteration 18341 / 24460) loss: 1.021586
(Epoch 15 / 20) train acc: 0.705000; val_acc: 0.659000
(Iteration 18361 / 24460) loss: 1.267448
(Iteration 18381 / 24460) loss: 0.877522
(Iteration 18401 / 24460) loss: 0.987045
(Iteration 18421 / 24460) loss: 0.920265
(Iteration 18441 / 24460) loss: 1.018046
(Iteration 18461 / 24460) loss: 1.119168
(Iteration 18481 / 24460) loss: 1.035933
(Iteration 18501 / 24460) loss: 1.230830
(Iteration 18521 / 24460) loss: 0.932665
(Iteration 18541 / 24460) loss: 1.100426
(Iteration 18561 / 24460) loss: 1.024486
(Iteration 18581 / 24460) loss: 1.207343
(Iteration 18601 / 24460) loss: 0.758447
(Iteration 18621 / 24460) loss: 1.066342
(Iteration 18641 / 24460) loss: 1.058629
(Iteration 18661 / 24460) loss: 0.918161
(Iteration 18681 / 24460) loss: 1.133171
(Iteration 18701 / 24460) loss: 0.927118
(Iteration 18721 / 24460) loss: 1.023816
(Iteration 18741 / 24460) loss: 1.028747
(Iteration 18761 / 24460) loss: 0.879278
(Iteration 18781 / 24460) loss: 0.983418
(Iteration 18801 / 24460) loss: 0.799960
(Iteration 18821 / 24460) loss: 1.112307
(Iteration 18841 / 24460) loss: 1.040959
(Iteration 18861 / 24460) loss: 1.104302
(Iteration 18881 / 24460) loss: 1.092357
(Iteration 18901 / 24460) loss: 1.294326
(Iteration 18921 / 24460) loss: 0.702518
(Iteration 18941 / 24460) loss: 1.150760
(Iteration 18961 / 24460) loss: 1.071607
(Iteration 18981 / 24460) loss: 0.844091
(Iteration 19001 / 24460) loss: 1.074861
(Iteration 19021 / 24460) loss: 1.209561
(Iteration 19041 / 24460) loss: 1.071704
(Iteration 19061 / 24460) loss: 1.059813
(Iteration 19081 / 24460) loss: 0.942126
(Iteration 19101 / 24460) loss: 0.940936
(Iteration 19121 / 24460) loss: 1.121654
(Iteration 19141 / 24460) loss: 0.967311
(Iteration 19161 / 24460) loss: 1.058338
(Iteration 19181 / 24460) loss: 1.104062
(Iteration 19201 / 24460) loss: 1.024871
(Iteration 19221 / 24460) loss: 1.086265
(Iteration 19241 / 24460) loss: 1.230465
(Iteration 19261 / 24460) loss: 0.773201
(Iteration 19281 / 24460) loss: 0.941570
(Iteration 19301 / 24460) loss: 0.818016
(Iteration 19321 / 24460) loss: 1.115881
(Iteration 19341 / 24460) loss: 0.877132
(Iteration 19361 / 24460) loss: 1.086820
```

```
(Iteration 19381 / 24460) loss: 0.995085
(Iteration 19401 / 24460) loss: 1.138334
(Iteration 19421 / 24460) loss: 0.798456
(Iteration 19441 / 24460) loss: 1.066905
(Iteration 19461 / 24460) loss: 0.917590
(Iteration 19481 / 24460) loss: 0.821619
(Iteration 19501 / 24460) loss: 0.847718
(Iteration 19521 / 24460) loss: 0.818222
(Iteration 19541 / 24460) loss: 0.941492
(Iteration 19561 / 24460) loss: 1.150525
(Epoch 16 / 20) train acc: 0.738000; val_acc: 0.663000
(Iteration 19581 / 24460) loss: 0.933560
(Iteration 19601 / 24460) loss: 0.932974
(Iteration 19621 / 24460) loss: 0.960941
(Iteration 19641 / 24460) loss: 1.010069
(Iteration 19661 / 24460) loss: 0.937096
(Iteration 19681 / 24460) loss: 1.108464
(Iteration 19701 / 24460) loss: 1.126899
(Iteration 19721 / 24460) loss: 1.153049
(Iteration 19741 / 24460) loss: 0.927695
(Iteration 19761 / 24460) loss: 1.140857
(Iteration 19781 / 24460) loss: 0.934053
(Iteration 19801 / 24460) loss: 0.924502
(Iteration 19821 / 24460) loss: 0.986542
(Iteration 19841 / 24460) loss: 1.110408
(Iteration 19861 / 24460) loss: 1.109872
(Iteration 19881 / 24460) loss: 0.820025
(Iteration 19901 / 24460) loss: 0.939441
(Iteration 19921 / 24460) loss: 1.096329
(Iteration 19941 / 24460) loss: 0.772496
(Iteration 19961 / 24460) loss: 0.783721
(Iteration 19981 / 24460) loss: 0.740935
(Iteration 20001 / 24460) loss: 0.850185
(Iteration 20021 / 24460) loss: 0.967168
(Iteration 20041 / 24460) loss: 1.026493
(Iteration 20061 / 24460) loss: 1.076235
(Iteration 20081 / 24460) loss: 0.683056
(Iteration 20101 / 24460) loss: 1.126611
(Iteration 20121 / 24460) loss: 1.043414
(Iteration 20141 / 24460) loss: 0.883252
(Iteration 20161 / 24460) loss: 1.119689
(Iteration 20181 / 24460) loss: 1.014996
(Iteration 20201 / 24460) loss: 0.917797
(Iteration 20221 / 24460) loss: 1.060991
(Iteration 20241 / 24460) loss: 1.023356
(Iteration 20261 / 24460) loss: 1.140276
(Iteration 20281 / 24460) loss: 1.073636
(Iteration 20301 / 24460) loss: 0.941061
(Iteration 20321 / 24460) loss: 1.152275
(Iteration 20341 / 24460) loss: 0.926957
(Iteration 20361 / 24460) loss: 1.088396
(Iteration 20381 / 24460) loss: 1.150701
(Iteration 20401 / 24460) loss: 0.877152
(Iteration 20421 / 24460) loss: 1.014255
(Iteration 20441 / 24460) loss: 1.087001
(Iteration 20461 / 24460) loss: 1.038355
(Iteration 20481 / 24460) loss: 0.827384
(Iteration 20501 / 24460) loss: 0.995229
(Iteration 20521 / 24460) loss: 1.035703
(Iteration 20541 / 24460) loss: 1.074615
(Iteration 20561 / 24460) loss: 1.138568
(Iteration 20581 / 24460) loss: 0.933145
(Iteration 20601 / 24460) loss: 1.007828
(Iteration 20621 / 24460) loss: 1.041655
(Iteration 20641 / 24460) loss: 1.016891
(Iteration 20661 / 24460) loss: 0.746718
(Iteration 20681 / 24460) loss: 1.116830
(Iteration 20701 / 24460) loss: 1.170004
(Iteration 20721 / 24460) loss: 1.099780
(Iteration 20741 / 24460) loss: 1.028325
(Iteration 20761 / 24460) loss: 0.834976
(Iteration 20781 / 24460) loss: 1.022568
(Epoch 17 / 20) train acc: 0.731000; val_acc: 0.640000
(Iteration 20801 / 24460) loss: 0.922046
(Iteration 20821 / 24460) loss: 1.174673
(Iteration 20841 / 24460) loss: 0.950158
(Iteration 20861 / 24460) loss: 0.768901
(Iteration 20881 / 24460) loss: 0.959578
(Iteration 20901 / 24460) loss: 1.329108
(Iteration 20921 / 24460) loss: 1.082409
(Iteration 20941 / 24460) loss: 1.202541
(Iteration 20961 / 24460) loss: 1.064210
(Iteration 20981 / 24460) loss: 1.060530
(Iteration 21001 / 24460) loss: 0.934925
(Iteration 21021 / 24460) loss: 0.950470
(Iteration 21041 / 24460) loss: 1.143485
(Iteration 21061 / 24460) loss: 1.077147
(Iteration 21081 / 24460) loss: 0.896111
(Iteration 21101 / 24460) loss: 0.940637
```

```
(Iteration 21121 / 24460) loss: 1.007318
(Iteration 21141 / 24460) loss: 1.023476
(Iteration 21161 / 24460) loss: 1.095456
(Iteration 21181 / 24460) loss: 0.950299
(Iteration 21201 / 24460) loss: 1.056925
(Iteration 21221 / 24460) loss: 1.069449
(Iteration 21241 / 24460) loss: 0.930335
(Iteration 21261 / 24460) loss: 1.018772
(Iteration 21281 / 24460) loss: 1.068299
(Iteration 21301 / 24460) loss: 1.035615
(Iteration 21321 / 24460) loss: 1.091287
(Iteration 21341 / 24460) loss: 0.963149
(Iteration 21361 / 24460) loss: 0.953681
(Iteration 21381 / 24460) loss: 1.288552
(Iteration 21401 / 24460) loss: 1.106612
(Iteration 21421 / 24460) loss: 0.925630
(Iteration 21441 / 24460) loss: 1.011891
(Iteration 21461 / 24460) loss: 0.965011
(Iteration 21481 / 24460) loss: 0.818149
(Iteration 21501 / 24460) loss: 1.033039
(Iteration 21521 / 24460) loss: 0.909476
(Iteration 21541 / 24460) loss: 1.001040
(Iteration 21561 / 24460) loss: 1.010531
(Iteration 21581 / 24460) loss: 1.032969
(Iteration 21601 / 24460) loss: 0.893847
(Iteration 21621 / 24460) loss: 0.910541
(Iteration 21641 / 24460) loss: 1.023789
(Iteration 21661 / 24460) loss: 0.952187
(Iteration 21681 / 24460) loss: 1.053918
(Iteration 21701 / 24460) loss: 0.739865
(Iteration 21721 / 24460) loss: 0.891961
(Iteration 21741 / 24460) loss: 0.972883
(Iteration 21761 / 24460) loss: 1.123074
(Iteration 21781 / 24460) loss: 1.097077
(Iteration 21801 / 24460) loss: 0.943170
(Iteration 21821 / 24460) loss: 1.067989
(Iteration 21841 / 24460) loss: 0.946845
(Iteration 21861 / 24460) loss: 0.883327
(Iteration 21881 / 24460) loss: 1.072923
(Iteration 21901 / 24460) loss: 0.947899
(Iteration 21921 / 24460) loss: 1.012109
(Iteration 21941 / 24460) loss: 1.148623
(Iteration 21961 / 24460) loss: 0.868122
(Iteration 21981 / 24460) loss: 0.925775
(Iteration 22001 / 24460) loss: 1.138566
(Epoch 18 / 20) train acc: 0.739000; val_acc: 0.655000
(Iteration 22021 / 24460) loss: 0.912006
(Iteration 22041 / 24460) loss: 0.811229
(Iteration 22061 / 24460) loss: 0.965356
(Iteration 22081 / 24460) loss: 0.924889
(Iteration 22101 / 24460) loss: 1.004426
(Iteration 22121 / 24460) loss: 1.018337
(Iteration 22141 / 24460) loss: 0.974854
(Iteration 22161 / 24460) loss: 1.189600
(Iteration 22181 / 24460) loss: 0.807168
(Iteration 22201 / 24460) loss: 0.913105
(Iteration 22221 / 24460) loss: 1.197313
(Iteration 22241 / 24460) loss: 0.967178
(Iteration 22261 / 24460) loss: 0.925176
(Iteration 22281 / 24460) loss: 0.987426
(Iteration 22301 / 24460) loss: 1.097462
(Iteration 22321 / 24460) loss: 0.977715
(Iteration 22341 / 24460) loss: 0.845273
(Iteration 22361 / 24460) loss: 1.209875
(Iteration 22381 / 24460) loss: 0.965679
(Iteration 22401 / 24460) loss: 0.981555
(Iteration 22421 / 24460) loss: 0.890439
(Iteration 22441 / 24460) loss: 1.123577
(Iteration 22461 / 24460) loss: 1.034118
(Iteration 22481 / 24460) loss: 0.913686
(Iteration 22501 / 24460) loss: 0.889586
(Iteration 22521 / 24460) loss: 1.036492
(Iteration 22541 / 24460) loss: 0.821065
(Iteration 22561 / 24460) loss: 0.814652
(Iteration 22581 / 24460) loss: 1.003390
(Iteration 22601 / 24460) loss: 0.918367
(Iteration 22621 / 24460) loss: 1.137092
(Iteration 22641 / 24460) loss: 1.051695
(Iteration 22661 / 24460) loss: 0.975818
(Iteration 22681 / 24460) loss: 0.818834
(Iteration 22701 / 24460) loss: 0.998922
(Iteration 22721 / 24460) loss: 1.039550
(Iteration 22741 / 24460) loss: 0.888914
(Iteration 22761 / 24460) loss: 0.968148
(Iteration 22781 / 24460) loss: 0.734471
(Iteration 22801 / 24460) loss: 0.788301
(Iteration 22821 / 24460) loss: 1.228171
(Iteration 22841 / 24460) loss: 1.036166
(Iteration 22861 / 24460) loss: 0.970328
```

```
(Iteration 22881 / 24460) loss: 1.068299
(Iteration 22901 / 24460) loss: 0.924555
(Iteration 22921 / 24460) loss: 1.044429
(Iteration 22941 / 24460) loss: 1.010795
(Iteration 22961 / 24460) loss: 1.064677
(Iteration 22981 / 24460) loss: 0.960362
(Iteration 23001 / 24460) loss: 0.818403
(Iteration 23021 / 24460) loss: 0.716439
(Iteration 23041 / 24460) loss: 1.070523
(Iteration 23061 / 24460) loss: 1.140401
(Iteration 23081 / 24460) loss: 1.080045
(Iteration 23101 / 24460) loss: 0.851951
(Iteration 23121 / 24460) loss: 0.831697
(Iteration 23141 / 24460) loss: 0.883531
(Iteration 23161 / 24460) loss: 1.261444
(Iteration 23181 / 24460) loss: 1.049552
(Iteration 23201 / 24460) loss: 0.904911
(Iteration 23221 / 24460) loss: 0.915234
(Epoch 19 / 20) train acc: 0.735000; val_acc: 0.651000
(Iteration 23241 / 24460) loss: 0.826024
(Iteration 23261 / 24460) loss: 1.014755
(Iteration 23281 / 24460) loss: 1.100332
(Iteration 23301 / 24460) loss: 1.076190
(Iteration 23321 / 24460) loss: 0.938992
(Iteration 23341 / 24460) loss: 0.908913
(Iteration 23361 / 24460) loss: 1.157090
(Iteration 23381 / 24460) loss: 0.867933
(Iteration 23401 / 24460) loss: 1.172429
(Iteration 23421 / 24460) loss: 1.057374
(Iteration 23441 / 24460) loss: 1.045948
(Iteration 23461 / 24460) loss: 0.863476
(Iteration 23481 / 24460) loss: 1.073397
(Iteration 23501 / 24460) loss: 1.167553
(Iteration 23521 / 24460) loss: 0.887552
(Iteration 23541 / 24460) loss: 0.881190
(Iteration 23561 / 24460) loss: 0.915276
(Iteration 23581 / 24460) loss: 1.083167
(Iteration 23601 / 24460) loss: 0.973142
(Iteration 23621 / 24460) loss: 0.939102
(Iteration 23641 / 24460) loss: 0.906110
(Iteration 23661 / 24460) loss: 1.033036
(Iteration 23681 / 24460) loss: 0.976433
(Iteration 23701 / 24460) loss: 0.842401
(Iteration 23721 / 24460) loss: 0.898538
(Iteration 23741 / 24460) loss: 1.107159
(Iteration 23761 / 24460) loss: 0.921357
(Iteration 23781 / 24460) loss: 0.987073
(Iteration 23801 / 24460) loss: 0.945137
(Iteration 23821 / 24460) loss: 0.801223
(Iteration 23841 / 24460) loss: 0.954122
(Iteration 23861 / 24460) loss: 1.047469
(Iteration 23881 / 24460) loss: 1.045736
(Iteration 23901 / 24460) loss: 1.012967
(Iteration 23921 / 24460) loss: 0.908905
(Iteration 23941 / 24460) loss: 0.834951
(Iteration 23961 / 24460) loss: 0.936099
(Iteration 23981 / 24460) loss: 1.073113
(Iteration 24001 / 24460) loss: 1.166132
(Iteration 24021 / 24460) loss: 1.221921
(Iteration 24041 / 24460) loss: 0.931706
(Iteration 24061 / 24460) loss: 1.238801
(Iteration 24081 / 24460) loss: 0.991921
(Iteration 24101 / 24460) loss: 1.008813
(Iteration 24121 / 24460) loss: 1.097185
(Iteration 24141 / 24460) loss: 0.880008
(Iteration 24161 / 24460) loss: 0.921586
(Iteration 24181 / 24460) loss: 0.920023
(Iteration 24201 / 24460) loss: 1.062129
(Iteration 24221 / 24460) loss: 0.970530
(Iteration 24241 / 24460) loss: 1.058508
(Iteration 24261 / 24460) loss: 0.894139
(Iteration 24281 / 24460) loss: 1.013689
(Iteration 24301 / 24460) loss: 1.112764
(Iteration 24321 / 24460) loss: 1.039305
(Iteration 24341 / 24460) loss: 0.895802
(Iteration 24361 / 24460) loss: 0.959915
(Iteration 24381 / 24460) loss: 1.231020
(Iteration 24401 / 24460) loss: 0.997188
(Iteration 24421 / 24460) loss: 0.961145
(Iteration 24441 / 24460) loss: 0.923600
(Epoch 20 / 20) train acc: 0.740000; val_acc: 0.661000
Best validation accuracy: 0.663
```

In [7]:

```
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
```

```python
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

from cs231n.data_utils import get_CIFAR10_data  # Assuming this is where your data loading is

data = get_CIFAR10_data()
for k in data.keys():
    print('{}: {} '.format(k, data[k].shape))
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


X_train = torch.from_numpy(data['X_train']).float().to(device)
y_train = torch.from_numpy(data['y_train']).long().to(device)
X_test = torch.from_numpy(data['X_test']).float().to(device)
y_test = torch.from_numpy(data['y_test']).long().to(device)

# Create TensorDatasets and DataLoaders
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = torch.utils.data.TensorDataset(X_test, y_test)
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# Define the 3-layer CNN with Batch Normalization
class ThreeLayerConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ThreeLayerConvNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, 2)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.bn4 = nn.BatchNorm1d(512)
        self.relu4 = nn.ReLU()
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
        x = self.pool3(self.relu3(self.bn3(self.conv3(x))))
        x = x.view(-1, 128 * 4 * 4)
        x = self.relu4(self.bn4(self.fc1(x)))
        x = self.fc2(x)
        return x

# Instantiate the model
net = ThreeLayerConvNet().to(device)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Training loop
num_epochs = 20
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 200 == 199:
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 200:.3f}')
            running_loss = 0.0

print('Finished Training')

# Evaluation
```

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 1000 test images: {100 * correct / total:.2f} %')
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
[1,   200] loss: 1.321
[1,   400] loss: 1.021
[1,   600] loss: 0.936
[2,   200] loss: 0.721
[2,   400] loss: 0.720
[2,   600] loss: 0.664
[3,   200] loss: 0.516
[3,   400] loss: 0.532
[3,   600] loss: 0.563
[4,   200] loss: 0.380
[4,   400] loss: 0.409
[4,   600] loss: 0.427
[5,   200] loss: 0.273
[5,   400] loss: 0.292
[5,   600] loss: 0.321
[6,   200] loss: 0.190
[6,   400] loss: 0.194
[6,   600] loss: 0.240
[7,   200] loss: 0.136
[7,   400] loss: 0.155
[7,   600] loss: 0.172
[8,   200] loss: 0.092
[8,   400] loss: 0.110
[8,   600] loss: 0.131
[9,   200] loss: 0.084
[9,   400] loss: 0.100
[9,   600] loss: 0.114
[10,   200] loss: 0.073
[10,   400] loss: 0.067
[10,   600] loss: 0.094
[11,   200] loss: 0.063
[11,   400] loss: 0.065
[11,   600] loss: 0.086
[12,   200] loss: 0.050
[12,   400] loss: 0.055
[12,   600] loss: 0.064
[13,   200] loss: 0.054
[13,   400] loss: 0.055
[13,   600] loss: 0.074
[14,   200] loss: 0.047
[14,   400] loss: 0.051
[14,   600] loss: 0.066
[15,   200] loss: 0.042
[15,   400] loss: 0.038
[15,   600] loss: 0.052
[16,   200] loss: 0.043
[16,   400] loss: 0.041
[16,   600] loss: 0.050
[17,   200] loss: 0.048
[17,   400] loss: 0.037
[17,   600] loss: 0.047
[18,   200] loss: 0.038
[18,   400] loss: 0.035
[18,   600] loss: 0.043
[19,   200] loss: 0.035
[19,   400] loss: 0.037
[19,   600] loss: 0.036
[20,   200] loss: 0.030
[20,   400] loss: 0.034
[20,   600] loss: 0.036
Finished Training
Accuracy of the network on the 1000 test images: 76.60 %
```