fully_connected_networks

May 15, 2024

1 ITE4052 Assignment 4-1: Fully-Connected Neural Networks and Dropout

- This material draws from EECS 498-007/598-005 (Justin Johnson)

Before we start, please put your name and HYID in following format Firstname LASTNAME, #00000000 // e.g.) Sukmin Yun, #12345678

Your Answer:
Junwoo Park, #2021006253

1.1 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment, same as Assignment 1. You'll need to rerun this setup code each time you start the notebook.

First, run this cell load the autoreload extension. This allows us to edit .py source files, and re-import them into the notebook for a seamless editing and debugging experience.

[1]:
```
%load_ext autoreload
%autoreload 2
```

1.1.1 Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Now recall the path in your Google Drive where you uploaded this notebook, fill it in below. If everything is working correctly then running the folowing cell should print the filenames from the assignment:

```
['convolutional_networks.ipynb', 'fully_connected_networks.ipynb', 'ite4052', 'convolutional_n
```

[3]:
```
import os

# TODO: Fill in the Google Drive path where you uploaded the assignment
# Example: If you create a ITE4052 folder and put all the files under A4␣
  ↪folder, then 'ITE4052/A4'
GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'ITE4052/A4'
GOOGLE_DRIVE_PATH = os.path.join('drive', 'My Drive',␣
  ↪GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
print(os.listdir(GOOGLE_DRIVE_PATH))
```

```
['makepdf.py', 'collectSubmission.sh', 'collect_submission.ipynb',
'a4_helper.py', 'ite4052', '__pycache__', 'best_two_layer_net.pth',
'best_overfit_five_layer_net.pth', 'fully_connected_networks.ipynb',
'overfit_deepconvnet.pth', 'one_minute_deepconvnet.pth',
'convolutional_networks.ipynb', 'convolutional_networks.py',
'fully_connected_networks.py']
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run th following cell to allow us to import from the .py files of this assignment. If it works correctly, it should print the message:

```
Hello from fully_connected_networks.py!
Hello from a4_helper.py!
```

as well as the last edit time for the file `fully_connected_networks.py`.

[4]:
```
import sys
sys.path.append(GOOGLE_DRIVE_PATH)

import time, os
os.environ["TZ"] = "US/Eastern"
time.tzset()

from fully_connected_networks import hello_fully_connected_networks
hello_fully_connected_networks()

from a4_helper import hello_helper
hello_helper()

fully_connected_networks_path = os.path.join(GOOGLE_DRIVE_PATH,␣
  ↪'fully_connected_networks.py')
fully_connected_networks_edit_time = time.ctime(os.path.
  ↪getmtime(fully_connected_networks_path))
print('fully_connected_networks.py last edited on %s' %␣
  ↪fully_connected_networks_edit_time)
```

```
Hello from fully_connected_networks.py!
Hello from a4_helper.py!
fully_connected_networks.py last edited on Wed May 15 11:21:15 2024
```

2 Data preprocessing

2.1 Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

[5]:
```
import ite4052
import torch
import torchvision
import matplotlib.pyplot as plt
import statistics
import random
import time
import math
%matplotlib inline

from ite4052 import reset_seed, Solver

plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['font.size'] = 16
```

Starting in this assignment, we will use the GPU to accelerate our computation. Run this cell to make sure you are using a GPU.

[6]:
```
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

2.2 Load the CIFAR-10 dataset

Then, we will first load the CIFAR-10 dataset. The utility function `ite4052.data.preprocess_cifar10()` returns the entire CIFAR-10 dataset as a set of six **Torch tensors** while also preprocessing the RGB images:

- `X_train` contains all training images (real numbers in the range $[0, 1]$)
- `y_train` contains all training labels (integers in the range $[0, 9]$)
- `X_val` contains all validation images
- `y_val` contains all validation labels
- `X_test` contains all test images
- `y_test` contains all test labels

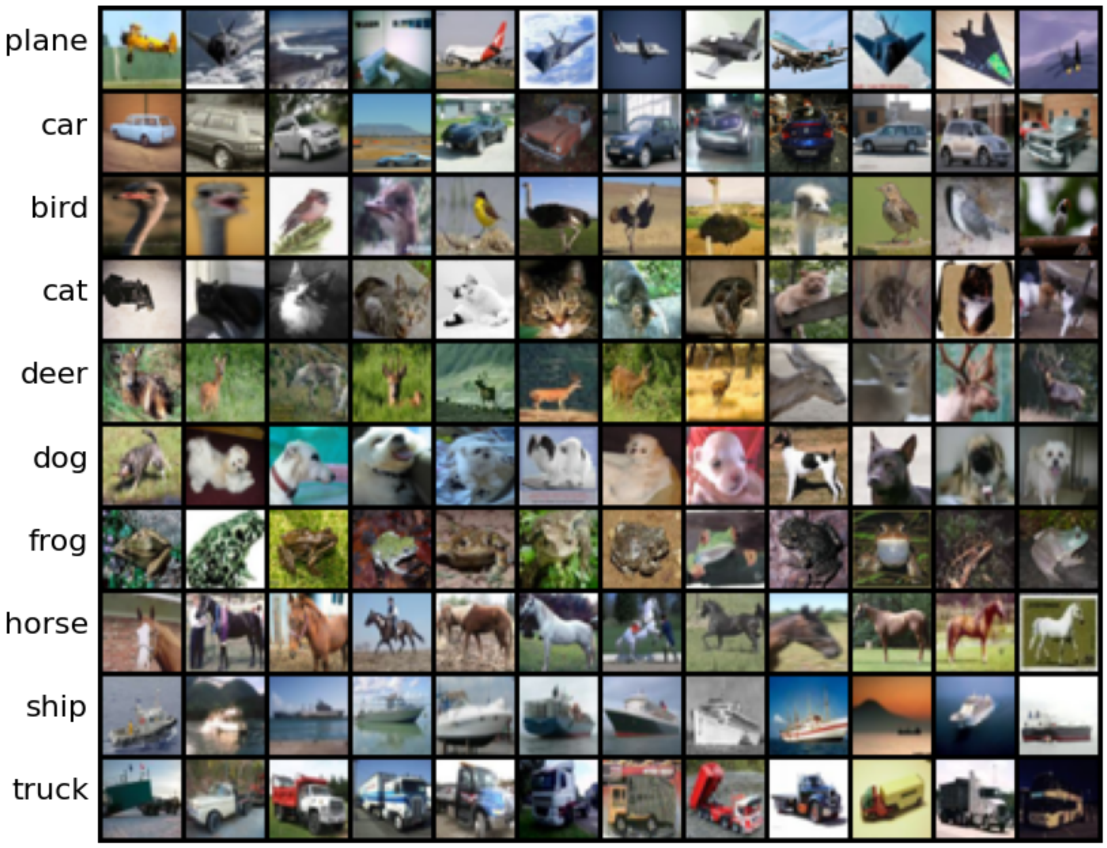
[7]:
```
# Invoke the above function to get our data.
import ite4052

ite4052.reset_seed(0)
data_dict = ite4052.data.preprocess_cifar10(cuda=True, dtype=torch.float64)
```

```
print('Train data shape: ', data_dict['X_train'].shape)
print('Train labels shape: ', data_dict['y_train'].shape)
print('Validation data shape: ', data_dict['X_val'].shape)
print('Validation labels shape: ', data_dict['y_val'].shape)
print('Test data shape: ', data_dict['X_test'].shape)
print('Test labels shape: ', data_dict['y_test'].shape)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./cifar-10-python.tar.gz
```

```
100%|      | 170498071/170498071 [00:04<00:00, 41550086.52it/s]
```

```
Extracting ./cifar-10-python.tar.gz to .
```



```
Train data shape:  torch.Size([40000, 3072])
Train labels shape:  torch.Size([40000])
Validation data shape:  torch.Size([10000, 3072])
Validation labels shape:  torch.Size([10000])
Test data shape:  torch.Size([10000, 3072])
Test labels shape:  torch.Size([10000])
```

3 Fully-connected neural networks

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer as a tool to more efficiently optimize deep networks.

To validate our implementation, we will compare the analytically computed gradients with numerical approximations of the gradient as done in previous assignments. You can inspect the numeric gradient function `compute_numeric_gradient` in `ite4052/grad.py`. Please note that we have updated the function to accept upstream gradients to allow us to debug intermediate layers easily. You can check the update description by running the code block below.

[8]:
```
help(ite4052.grad.compute_numeric_gradient)
```

```
Help on function compute_numeric_gradient in module ite4052.grad:

compute_numeric_gradient(f, x, dLdf=None, h=1e-07)
    Compute the numeric gradient of f at x using a finite differences
    approximation. We use the centered difference:

    df    f(x + h) - f(x - h)
    -- ~= -------------------
    dx           2 * h

    Function can also expand this easily to intermediate layers using the
    chain rule:

    dL   df   dL
    -- = -- * --
    dx   dx   df

    Inputs:
    - f: A function that inputs a torch tensor and returns a torch scalar
    - x: A torch tensor giving the point at which to compute the gradient
    - dLdf: optional upstream gradient for intermediate layers
    - h: epsilon used in the finite difference calculation
    Returns:
    - grad: A tensor of the same shape as x giving the gradient of f at x
```

4 Linear layer

For each layer we implement, we will define a class with two static methods `forward` and `backward`. The class structure is currently provided in `fully_connected_layers.py`, you will be implementing both the `forward` and `backward` methods.

4.1 Linear layer: forward

Implement the `Linear.forward` function in `fully_connected_layers.py`. Once you are done you can test your implementaion by running the next cell. You should see errors less than `1e-7`.

[9]:
```
from fully_connected_networks import Linear

# Test the Linear.forward function
num_inputs = 2
input_shape = torch.tensor((4, 5, 6))
```

```
output_dim = 3

input_size = num_inputs * torch.prod(input_shape)
weight_size = output_dim * torch.prod(input_shape)

x = torch.linspace(-0.1, 0.5, steps=input_size, dtype=torch.float64,␣
 ↪device='cuda')
w = torch.linspace(-0.2, 0.3, steps=weight_size, dtype=torch.float64,␣
 ↪device='cuda')
b = torch.linspace(-0.3, 0.1, steps=output_dim, dtype=torch.float64,␣
 ↪device='cuda')
x = x.reshape(num_inputs, *input_shape)
w = w.reshape(torch.prod(input_shape), output_dim)

out, _ = Linear.forward(x, w, b)
correct_out = torch.tensor([[1.49834984, 1.70660150, 1.91485316],
                            [3.25553226, 3.51413301, 3.77273372]]
                            ).double().cuda()

print('Testing Linear.forward function:')
print('difference: ', ite4052.grad.rel_error(out, correct_out))
```

```
Testing Linear.forward function:
difference:  3.683042917976506e-08
```

4.2 Linear layer: backward

Now implement the `Linear.backward` function and test your implementation using numeric gradient checking.

Run the following to test your implementation of `Linear.backward`. You should see errors less than 1e-7.

[10]:
```
from fully_connected_networks import Linear

# Test the Linear.backward function
reset_seed(0)
x = torch.randn(10, 2, 3, dtype=torch.float64, device='cuda')
w = torch.randn(6, 5, dtype=torch.float64, device='cuda')
b = torch.randn(5, dtype=torch.float64, device='cuda')
dout = torch.randn(10, 5, dtype=torch.float64, device='cuda')

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: Linear.forward(x, w,␣
 ↪b)[0], x, dout)
dw_num = ite4052.grad.compute_numeric_gradient(lambda w: Linear.forward(x, w,␣
 ↪b)[0], w, dout)
db_num = ite4052.grad.compute_numeric_gradient(lambda b: Linear.forward(x, w,␣
 ↪b)[0], b, dout)
```

```
_, cache = Linear.forward(x, w, b)
dx, dw, db = Linear.backward(dout, cache)

# The error should be around e-10 or less
print('Testing Linear.backward function:')
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
print('dw error: ', ite4052.grad.rel_error(dw_num, dw))
print('db error: ', ite4052.grad.rel_error(db_num, db))
```

```
Testing Linear.backward function:
dx error:  5.221943563709987e-10
dw error:  3.498388787266994e-10
db error:  5.373171200544344e-10
```

5 ReLU activation

We will now implement the ReLU nonlinearity. As above, we will define a class with two empty static methods, and implement them in upcoming cells. The class structure can be found in `fully_connected_networks.py`

5.1 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `ReLU.forward` function. You **should not** change the input tensor with an in-place operation.

Run the following to test your implementation of the ReLU forward pass. Your errors should be less than `1e-7`.

[11]:
```
from fully_connected_networks import ReLU

reset_seed(0)
x = torch.linspace(-0.5, 0.5, steps=12, dtype=torch.float64, device='cuda')
x = x.reshape(3, 4)

out, _ = ReLU.forward(x)
correct_out = torch.tensor([[ 0.,          0.,          0.,          0.,        ␣
 ↪],
                            [ 0.,          0.,          0.04545455,  0.
 ↪13636364,],
                            [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ␣
 ↪]],
                            dtype=torch.float64,
                            device='cuda')

# Compare your output with ours. The error should be on the order of e-8
print('Testing ReLU.forward function:')
print('difference: ', ite4052.grad.rel_error(out, correct_out))
```

```
Testing ReLU.forward function:
difference:  4.5454545613554664e-09
```

5.2 ReLU activation: backward

Now implement the backward pass for the ReLU activation function.

Again, you should not change the input tensor with an in-place operation.

Run the following to test your implementation of ReLU.backward. Your errors should be less than 1e-8.

[12]:
```
from fully_connected_networks import ReLU

reset_seed(0)
x = torch.randn(10, 10, dtype=torch.float64, device='cuda')
dout = torch.randn(*x.shape, dtype=torch.float64, device='cuda')

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: ReLU.forward(x)[0], x,␣
  ↪dout)

_, cache = ReLU.forward(x)
dx = ReLU.backward(dout, cache)

# The error should be on the order of e-12
print('Testing ReLU.backward function:')
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
```

```
Testing ReLU.backward function:
dx error:  2.6317796097761553e-10
```

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, linear layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define a convenience layer.

This also shows how our layer abstraction allows us to implement new layers by composing existing layer implementations. This is a powerful mechanism for structuring deep learning code in a modular fashion.

For now take a look at the forward and backward functions in Linear_ReLU, and run the following to numerically gradient check the backward pass.

Run the following to test the implementation of the Linear_ReLU layer using numeric gradient checking. You should see errors less than 1e-8

[13]:
```
from fully_connected_networks import Linear_ReLU

reset_seed(0)
```

```
x = torch.randn(2, 3, 4, dtype=torch.float64, device='cuda')
w = torch.randn(12, 10, dtype=torch.float64, device='cuda')
b = torch.randn(10, dtype=torch.float64, device='cuda')
dout = torch.randn(2, 10, dtype=torch.float64, device='cuda')

out, cache = Linear_ReLU.forward(x, w, b)
dx, dw, db = Linear_ReLU.backward(dout, cache)

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: Linear_ReLU.forward(x,␣
  ↪w, b)[0], x, dout)
dw_num = ite4052.grad.compute_numeric_gradient(lambda w: Linear_ReLU.forward(x,␣
  ↪w, b)[0], w, dout)
db_num = ite4052.grad.compute_numeric_gradient(lambda b: Linear_ReLU.forward(x,␣
  ↪w, b)[0], b, dout)

# Relative error should be around e-8 or less
print('Testing Linear_ReLU.forward and Linear_ReLU.backward:')
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
print('dw error: ', ite4052.grad.rel_error(dw_num, dw))
print('db error: ', ite4052.grad.rel_error(db_num, db))
```

```
Testing Linear_ReLU.forward and Linear_ReLU.backward:
dx error:  1.210759699545244e-09
dw error:  7.462948482161807e-10
db error:  8.915028842081707e-10
```

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free in `helper_functions.py`. You should still make sure you understand how they work by looking at the implementations. We can first verify our implementations.

Run the following to perform numeric gradient checking on the two loss functions. You should see errors less than `1e-7` for svm_loss and `1e-6` for softmax_loss.

[14]:
```
from a4_helper import svm_loss, softmax_loss

reset_seed(0)
num_classes, num_inputs = 10, 50
x = 0.001 * torch.randn(num_inputs, num_classes, dtype=torch.float64,␣
  ↪device='cuda')
y = torch.randint(num_classes, size=(num_inputs,), dtype=torch.int64,␣
  ↪device='cuda')

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: svm_loss(x, y)[0], x)
loss, dx = svm_loss(x, y)
```

```
# Test svm_loss function. Loss should be around 9 and dx error should be around␣
↪the order of e-9
print('Testing svm_loss:')
print('loss: ', loss.item())
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: softmax_loss(x, y)[0],␣
↪x)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should␣
↪be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss.item())
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  9.000430792478463
dx error:  2.0074935157654598e-08

Testing softmax_loss:
loss:  2.3026286102347924
dx error:  1.0417990899757076e-07
```

8 Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API.

Once you have finished implementing the forward and backward passes of your two-layer net, run the following to test your implementation:

[15]:
```
from fully_connected_networks import TwoLayerNet
from a4_helper import svm_loss, softmax_loss

reset_seed(0)
N, D, H, C = 3, 5, 50, 7
X = torch.randn(N, D, dtype=torch.float64, device='cuda')
y = torch.randint(C, size=(N,), dtype=torch.int64, device='cuda')

std = 1e-3
model = TwoLayerNet(
          input_dim=D,
          hidden_dim=H,
          num_classes=C,
          weight_scale=std,
          dtype=torch.float64,
          device='cuda'
        )

print('Testing initialization ... ')
W1_std = torch.abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = torch.abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert torch.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert torch.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = torch.linspace(-0.7, 0.3, steps=D * H, dtype=torch.
 ↪float64, device='cuda').reshape(D, H)
model.params['b1'] = torch.linspace(-0.1, 0.9, steps=H, dtype=torch.float64,␣
 ↪device='cuda')
model.params['W2'] = torch.linspace(-0.3, 0.4, steps=H * C, dtype=torch.
 ↪float64, device='cuda').reshape(H, C)
model.params['b2'] = torch.linspace(-0.9, 0.1, steps=C, dtype=torch.float64,␣
 ↪device='cuda')
X = torch.linspace(-5.5, 4.5, steps=N * D, dtype=torch.float64, device='cuda').
 ↪reshape(D, N).t()
scores = model.loss(X)
correct_scores = torch.tensor(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]],
    dtype=torch.float64, device='cuda')
scores_diff = torch.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = torch.tensor([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'
```

```
model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 49.719461034881775
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-6 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = ite4052.grad.compute_numeric_gradient(f, model.params[name])
    print('%s relative error: %.2e' % (name, ite4052.grad.rel_error(grad_num,␣
 ↪grads[name])))
```

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 2.94e-07
W2 relative error: 1.65e-09
b1 relative error: 1.01e-06
b2 relative error: 4.63e-09
Running numeric gradient check with reg =  0.7
W1 relative error: 2.70e-08
W2 relative error: 9.86e-09
b1 relative error: 2.28e-06
b2 relative error: 2.90e-08
```

9 Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

Read through `help(Solver)` to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

```
[16]: print(help(Solver))
```

```
Help on class Solver in module ite4052.solver:

class Solver(builtins.object)
 |  Solver(model, data, **kwargs)
 |
 |  A Solver encapsulates all the logic necessary for training classification
 |  models. The Solver performs stochastic gradient descent using different
 |  update rules.
 |  The solver accepts both training and validation data and labels so it can
 |  periodically check classification accuracy on both training and validation
 |  data to watch out for overfitting.
 |  To train a model, you will first construct a Solver instance, passing the
 |  model, dataset, and various options (learning rate, batch size, etc) to the
 |  constructor. You will then call the train() method to run the optimization
 |  procedure and train the model.
 |  After the train() method returns, model.params will contain the parameters
 |  that performed best on the validation set over the course of training.
 |  In addition, the instance variable solver.loss_history will contain a list
 |  of all losses encountered during training and the instance variables
 |  solver.train_acc_history and solver.val_acc_history will be lists of the
 |  accuracies of the model on the training and validation set at each epoch.
 |  Example usage might look something like this:
 |  data = {
 |    'X_train': # training data
 |    'y_train': # training labels
 |    'X_val': # validation data
 |    'y_val': # validation labels
 |  }
 |  model = MyAwesomeModel(hidden_size=100, reg=10)
 |  solver = Solver(model, data,
 |          update_rule=sgd,
 |          optim_config={
 |            'learning_rate': 1e-3,
 |          },
 |          lr_decay=0.95,
 |          num_epochs=10, batch_size=100,
 |          print_every=100,
 |          device='cuda')
 |  solver.train()
 |  A Solver works on a model object that must conform to the following API:
 |  - model.params must be a dictionary mapping string parameter names to torch
 |    tensors containing parameter values.
 |  - model.loss(X, y) must be a function that computes training-time loss and
 |    gradients, and test-time classification scores, with the following inputs
 |    and outputs:
 |    Inputs:
 |    - X: Array giving a minibatch of input data of shape (N, d_1, …, d_k)
 |    - y: Array of labels, of shape (N,) giving labels for X where y[i] is the
 |    label for X[i].
 |    Returns:
 |    If y is None, run a test-time forward pass and return:
 |    - scores: Array of shape (N, C) giving classification scores for X where
 |    scores[i, c] gives the score of class c for X[i].
 |    If y is not None, run a training time forward and backward pass and
 |    return a tuple of:
 |    - loss: Scalar giving the loss
 |    - grads: Dictionary with the same keys as self.params mapping parameter
 |    names to gradients of the loss with respect to those parameters.
 |    - device: device to use for computation. 'cpu' or 'cuda'
 |
 |  Methods defined here:
 |
 |  __init__(self, model, data, **kwargs)
 |      Construct a new Solver instance.
 |      Required arguments:
 |      - model: A model object conforming to the API described above
 |      - data: A dictionary of training and validation data containing:
 |        'X_train': Array, shape (N_train, d_1, …, d_k) of training images
 |        'X_val': Array, shape (N_val, d_1, …, d_k) of validation images
 |        'y_train': Array, shape (N_train,) of labels for training images
 |        'y_val': Array, shape (N_val,) of labels for validation images
 |      Optional arguments:
 |      - update_rule: A function of an update rule. Default is sgd.
 |      - optim_config: A dictionary containing hyperparameters that will be
 |        passed to the chosen update rule. Each update rule requires different
 |        hyperparameters but all update rules require a
 |        'learning_rate' parameter so that should always be present.
 |      - lr_decay: A scalar for learning rate decay; after each epoch the
 |        learning rate is multiplied by this value.
 |      - batch_size: Size of minibatches used to compute loss and gradient
 |        during training.
 |      - num_epochs: The number of epochs to run for during training.
 |      - print_every: Integer; training losses will be printed every
 |        print_every iterations.
 |      - print_acc_every: We will print the accuracy every
 |        print_acc_every epochs.
 |      - verbose: Boolean; if set to false then no output will be printed
 |        during training.
 |      - num_train_samples: Number of training samples used to check training
 |        accuracy; default is 1000; set to None to use entire training set.
 |      - num_val_samples: Number of validation samples to use to check val
 |        accuracy; default is None, which uses the entire validation set.
 |      - checkpoint_name: If not None, then save model checkpoints here every
 |        epoch.
 |
 |  check_accuracy(self, X, y, num_samples=None, batch_size=100)
 |      Check accuracy of the model on the provided data.
 |      Inputs:
 |      - X: Array of data, of shape (N, d_1, …, d_k)
 |      - y: Array of labels, of shape (N,)
 |      - num_samples: If not None, subsample the data and only test the model
 |        on num_samples datapoints.
```

```
 |      - batch_size: Split X and y into batches of this size to avoid using
 |        too much memory.
 |      Returns:
 |      - acc: Scalar giving the fraction of instances that were correctly
 |        classified by the model.
 |
 |  train(self, time_limit=None, return_best_params=True)
 |      Run optimization to train the model.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  sgd(w, dw, config=None)
 |      Performs vanilla stochastic gradient descent.
 |      config format:
 |      - learning_rate: Scalar learning rate.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

None
```

Use the Solver classe to create a solver instance that trains a TwoLayerNet to achieve at least 50% performance on the validation set.

Implement `create_solver_instance` in `fully_connected_networks.py` to return a solver instance. Make sure to initialize the Solver instance with the input device.

[17]:
```
from fully_connected_networks import create_solver_instance

reset_seed(0)

# Create a solver instance that achieves 50% performance on the validation set
solver = create_solver_instance(data_dict=data_dict, dtype=torch.float64,␣
 ↪device='cuda')
solver.train()
```

```
(Time 0.04 sec; Iteration 1 / 8000) loss: 2.302603
(Epoch 0 / 20) train acc: 0.096000; val_acc: 0.092200
(Time 0.40 sec; Iteration 11 / 8000) loss: 2.302131
(Time 0.44 sec; Iteration 21 / 8000) loss: 2.301974
(Time 0.49 sec; Iteration 31 / 8000) loss: 2.302352
(Time 0.53 sec; Iteration 41 / 8000) loss: 2.297661
```

```
(Time 0.57 sec; Iteration 51 / 8000) loss: 2.283626
(Time 0.62 sec; Iteration 61 / 8000) loss: 2.268614
(Time 0.66 sec; Iteration 71 / 8000) loss: 2.212975
(Time 0.70 sec; Iteration 81 / 8000) loss: 2.185241
(Time 0.74 sec; Iteration 91 / 8000) loss: 2.173307
(Time 0.79 sec; Iteration 101 / 8000) loss: 2.183683
(Time 0.83 sec; Iteration 111 / 8000) loss: 2.091396
(Time 0.87 sec; Iteration 121 / 8000) loss: 2.059497
(Time 0.91 sec; Iteration 131 / 8000) loss: 2.073542
(Time 0.96 sec; Iteration 141 / 8000) loss: 2.043060
(Time 1.00 sec; Iteration 151 / 8000) loss: 2.057305
(Time 1.05 sec; Iteration 161 / 8000) loss: 2.060444
(Time 1.09 sec; Iteration 171 / 8000) loss: 1.969404
(Time 1.13 sec; Iteration 181 / 8000) loss: 2.018749
(Time 1.17 sec; Iteration 191 / 8000) loss: 1.918732
(Time 1.22 sec; Iteration 201 / 8000) loss: 1.925371
(Time 1.26 sec; Iteration 211 / 8000) loss: 1.986776
(Time 1.30 sec; Iteration 221 / 8000) loss: 2.040569
(Time 1.34 sec; Iteration 231 / 8000) loss: 1.862027
(Time 1.38 sec; Iteration 241 / 8000) loss: 1.882783
(Time 1.43 sec; Iteration 251 / 8000) loss: 1.999209
(Time 1.47 sec; Iteration 261 / 8000) loss: 1.881364
(Time 1.52 sec; Iteration 271 / 8000) loss: 1.869433
(Time 1.57 sec; Iteration 281 / 8000) loss: 1.920360
(Time 1.62 sec; Iteration 291 / 8000) loss: 1.734718
(Time 1.67 sec; Iteration 301 / 8000) loss: 1.746224
(Time 1.72 sec; Iteration 311 / 8000) loss: 1.827956
(Time 1.77 sec; Iteration 321 / 8000) loss: 1.936580
(Time 1.82 sec; Iteration 331 / 8000) loss: 1.991148
(Time 1.87 sec; Iteration 341 / 8000) loss: 1.981817
(Time 1.92 sec; Iteration 351 / 8000) loss: 1.872227
(Time 1.96 sec; Iteration 361 / 8000) loss: 1.688998
(Time 2.01 sec; Iteration 371 / 8000) loss: 1.839584
(Time 2.06 sec; Iteration 381 / 8000) loss: 1.810971
(Time 2.11 sec; Iteration 391 / 8000) loss: 1.808983
(Epoch 1 / 20) train acc: 0.361000; val_acc: 0.359000
(Time 2.27 sec; Iteration 401 / 8000) loss: 1.866122
(Time 2.32 sec; Iteration 411 / 8000) loss: 1.905465
(Time 2.37 sec; Iteration 421 / 8000) loss: 1.730916
(Time 2.41 sec; Iteration 431 / 8000) loss: 1.752849
(Time 2.46 sec; Iteration 441 / 8000) loss: 1.692051
(Time 2.51 sec; Iteration 451 / 8000) loss: 1.627449
(Time 2.57 sec; Iteration 461 / 8000) loss: 1.766686
(Time 2.62 sec; Iteration 471 / 8000) loss: 1.754263
(Time 2.67 sec; Iteration 481 / 8000) loss: 1.680517
(Time 2.72 sec; Iteration 491 / 8000) loss: 1.819147
(Time 2.77 sec; Iteration 501 / 8000) loss: 1.848964
(Time 2.82 sec; Iteration 511 / 8000) loss: 1.664147
```

```
(Time 2.87 sec; Iteration 521 / 8000) loss: 1.861571
(Time 2.92 sec; Iteration 531 / 8000) loss: 1.566806
(Time 2.97 sec; Iteration 541 / 8000) loss: 1.747006
(Time 3.02 sec; Iteration 551 / 8000) loss: 1.634745
(Time 3.07 sec; Iteration 561 / 8000) loss: 1.684015
(Time 3.11 sec; Iteration 571 / 8000) loss: 1.847447
(Time 3.16 sec; Iteration 581 / 8000) loss: 1.796701
(Time 3.21 sec; Iteration 591 / 8000) loss: 1.698253
(Time 3.26 sec; Iteration 601 / 8000) loss: 1.626178
(Time 3.31 sec; Iteration 611 / 8000) loss: 1.793206
(Time 3.36 sec; Iteration 621 / 8000) loss: 1.764089
(Time 3.41 sec; Iteration 631 / 8000) loss: 1.776800
(Time 3.46 sec; Iteration 641 / 8000) loss: 1.736117
(Time 3.50 sec; Iteration 651 / 8000) loss: 1.686904
(Time 3.55 sec; Iteration 661 / 8000) loss: 1.697896
(Time 3.60 sec; Iteration 671 / 8000) loss: 1.786791
(Time 3.65 sec; Iteration 681 / 8000) loss: 1.563450
(Time 3.71 sec; Iteration 691 / 8000) loss: 1.567511
(Time 3.76 sec; Iteration 701 / 8000) loss: 1.760979
(Time 3.81 sec; Iteration 711 / 8000) loss: 1.665903
(Time 3.85 sec; Iteration 721 / 8000) loss: 1.715495
(Time 3.91 sec; Iteration 731 / 8000) loss: 1.478780
(Time 3.96 sec; Iteration 741 / 8000) loss: 1.593218
(Time 4.01 sec; Iteration 751 / 8000) loss: 1.464391
(Time 4.07 sec; Iteration 761 / 8000) loss: 1.576131
(Time 4.12 sec; Iteration 771 / 8000) loss: 1.585751
(Time 4.17 sec; Iteration 781 / 8000) loss: 1.493866
(Time 4.23 sec; Iteration 791 / 8000) loss: 1.659011
(Epoch 2 / 20) train acc: 0.429000; val_acc: 0.416400
(Time 4.39 sec; Iteration 801 / 8000) loss: 1.522012
(Time 4.44 sec; Iteration 811 / 8000) loss: 1.524754
(Time 4.49 sec; Iteration 821 / 8000) loss: 1.693718
(Time 4.54 sec; Iteration 831 / 8000) loss: 1.619641
(Time 4.60 sec; Iteration 841 / 8000) loss: 1.661180
(Time 4.65 sec; Iteration 851 / 8000) loss: 1.731999
(Time 4.71 sec; Iteration 861 / 8000) loss: 1.388202
(Time 4.76 sec; Iteration 871 / 8000) loss: 1.494590
(Time 4.81 sec; Iteration 881 / 8000) loss: 1.719062
(Time 4.86 sec; Iteration 891 / 8000) loss: 1.696230
(Time 4.92 sec; Iteration 901 / 8000) loss: 1.720399
(Time 4.97 sec; Iteration 911 / 8000) loss: 1.634952
(Time 5.02 sec; Iteration 921 / 8000) loss: 1.641377
(Time 5.07 sec; Iteration 931 / 8000) loss: 1.613847
(Time 5.11 sec; Iteration 941 / 8000) loss: 1.416605
(Time 5.16 sec; Iteration 951 / 8000) loss: 1.493810
(Time 5.20 sec; Iteration 961 / 8000) loss: 1.609787
(Time 5.24 sec; Iteration 971 / 8000) loss: 1.434485
(Time 5.29 sec; Iteration 981 / 8000) loss: 1.673444
```

```
(Time 5.33 sec; Iteration 991 / 8000) loss: 1.492794
(Time 5.37 sec; Iteration 1001 / 8000) loss: 1.417730
(Time 5.41 sec; Iteration 1011 / 8000) loss: 1.515685
(Time 5.46 sec; Iteration 1021 / 8000) loss: 1.690853
(Time 5.50 sec; Iteration 1031 / 8000) loss: 1.463434
(Time 5.54 sec; Iteration 1041 / 8000) loss: 1.534649
(Time 5.59 sec; Iteration 1051 / 8000) loss: 1.518368
(Time 5.63 sec; Iteration 1061 / 8000) loss: 1.572649
(Time 5.67 sec; Iteration 1071 / 8000) loss: 1.496677
(Time 5.72 sec; Iteration 1081 / 8000) loss: 1.467277
(Time 5.76 sec; Iteration 1091 / 8000) loss: 1.411377
(Time 5.81 sec; Iteration 1101 / 8000) loss: 1.508326
(Time 5.85 sec; Iteration 1111 / 8000) loss: 1.565717
(Time 5.89 sec; Iteration 1121 / 8000) loss: 1.502134
(Time 5.93 sec; Iteration 1131 / 8000) loss: 1.524817
(Time 5.98 sec; Iteration 1141 / 8000) loss: 1.462300
(Time 6.02 sec; Iteration 1151 / 8000) loss: 1.596607
(Time 6.06 sec; Iteration 1161 / 8000) loss: 1.412042
(Time 6.11 sec; Iteration 1171 / 8000) loss: 1.525309
(Time 6.15 sec; Iteration 1181 / 8000) loss: 1.389256
(Time 6.19 sec; Iteration 1191 / 8000) loss: 1.458935
(Epoch 3 / 20) train acc: 0.447000; val_acc: 0.441400
(Time 6.34 sec; Iteration 1201 / 8000) loss: 1.573176
(Time 6.38 sec; Iteration 1211 / 8000) loss: 1.467197
(Time 6.43 sec; Iteration 1221 / 8000) loss: 1.607835
(Time 6.47 sec; Iteration 1231 / 8000) loss: 1.537023
(Time 6.52 sec; Iteration 1241 / 8000) loss: 1.591737
(Time 6.56 sec; Iteration 1251 / 8000) loss: 1.525906
(Time 6.60 sec; Iteration 1261 / 8000) loss: 1.488748
(Time 6.65 sec; Iteration 1271 / 8000) loss: 1.342857
(Time 6.69 sec; Iteration 1281 / 8000) loss: 1.487766
(Time 6.74 sec; Iteration 1291 / 8000) loss: 1.296228
(Time 6.79 sec; Iteration 1301 / 8000) loss: 1.415928
(Time 6.83 sec; Iteration 1311 / 8000) loss: 1.487382
(Time 6.87 sec; Iteration 1321 / 8000) loss: 1.467810
(Time 6.92 sec; Iteration 1331 / 8000) loss: 1.560782
(Time 6.96 sec; Iteration 1341 / 8000) loss: 1.538561
(Time 7.01 sec; Iteration 1351 / 8000) loss: 1.659575
(Time 7.05 sec; Iteration 1361 / 8000) loss: 1.516713
(Time 7.09 sec; Iteration 1371 / 8000) loss: 1.435643
(Time 7.13 sec; Iteration 1381 / 8000) loss: 1.490255
(Time 7.18 sec; Iteration 1391 / 8000) loss: 1.615745
(Time 7.22 sec; Iteration 1401 / 8000) loss: 1.681612
(Time 7.26 sec; Iteration 1411 / 8000) loss: 1.311217
(Time 7.31 sec; Iteration 1421 / 8000) loss: 1.531865
(Time 7.35 sec; Iteration 1431 / 8000) loss: 1.448577
(Time 7.39 sec; Iteration 1441 / 8000) loss: 1.437073
(Time 7.43 sec; Iteration 1451 / 8000) loss: 1.616219
```

```
(Time 7.48 sec; Iteration 1461 / 8000) loss: 1.583338
(Time 7.52 sec; Iteration 1471 / 8000) loss: 1.532556
(Time 7.56 sec; Iteration 1481 / 8000) loss: 1.563625
(Time 7.60 sec; Iteration 1491 / 8000) loss: 1.614664
(Time 7.65 sec; Iteration 1501 / 8000) loss: 1.386953
(Time 7.69 sec; Iteration 1511 / 8000) loss: 1.421970
(Time 7.73 sec; Iteration 1521 / 8000) loss: 1.266908
(Time 7.78 sec; Iteration 1531 / 8000) loss: 1.407091
(Time 7.82 sec; Iteration 1541 / 8000) loss: 1.641110
(Time 7.87 sec; Iteration 1551 / 8000) loss: 1.633930
(Time 7.91 sec; Iteration 1561 / 8000) loss: 1.402136
(Time 7.95 sec; Iteration 1571 / 8000) loss: 1.354367
(Time 7.99 sec; Iteration 1581 / 8000) loss: 1.241307
(Time 8.04 sec; Iteration 1591 / 8000) loss: 1.481929
(Epoch 4 / 20) train acc: 0.463000; val_acc: 0.459100
(Time 8.19 sec; Iteration 1601 / 8000) loss: 1.521766
(Time 8.23 sec; Iteration 1611 / 8000) loss: 1.242913
(Time 8.27 sec; Iteration 1621 / 8000) loss: 1.367333
(Time 8.32 sec; Iteration 1631 / 8000) loss: 1.551271
(Time 8.36 sec; Iteration 1641 / 8000) loss: 1.547451
(Time 8.40 sec; Iteration 1651 / 8000) loss: 1.404027
(Time 8.44 sec; Iteration 1661 / 8000) loss: 1.451265
(Time 8.49 sec; Iteration 1671 / 8000) loss: 1.512043
(Time 8.53 sec; Iteration 1681 / 8000) loss: 1.411674
(Time 8.57 sec; Iteration 1691 / 8000) loss: 1.390853
(Time 8.62 sec; Iteration 1701 / 8000) loss: 1.335102
(Time 8.66 sec; Iteration 1711 / 8000) loss: 1.486694
(Time 8.70 sec; Iteration 1721 / 8000) loss: 1.555205
(Time 8.75 sec; Iteration 1731 / 8000) loss: 1.563746
(Time 8.79 sec; Iteration 1741 / 8000) loss: 1.477284
(Time 8.84 sec; Iteration 1751 / 8000) loss: 1.505864
(Time 8.88 sec; Iteration 1761 / 8000) loss: 1.271154
(Time 8.92 sec; Iteration 1771 / 8000) loss: 1.525193
(Time 8.97 sec; Iteration 1781 / 8000) loss: 1.389323
(Time 9.01 sec; Iteration 1791 / 8000) loss: 1.468372
(Time 9.05 sec; Iteration 1801 / 8000) loss: 1.417168
(Time 9.10 sec; Iteration 1811 / 8000) loss: 1.416766
(Time 9.14 sec; Iteration 1821 / 8000) loss: 1.508289
(Time 9.18 sec; Iteration 1831 / 8000) loss: 1.426444
(Time 9.22 sec; Iteration 1841 / 8000) loss: 1.395155
(Time 9.26 sec; Iteration 1851 / 8000) loss: 1.336724
(Time 9.31 sec; Iteration 1861 / 8000) loss: 1.288793
(Time 9.35 sec; Iteration 1871 / 8000) loss: 1.305221
(Time 9.39 sec; Iteration 1881 / 8000) loss: 1.337113
(Time 9.44 sec; Iteration 1891 / 8000) loss: 1.498912
(Time 9.48 sec; Iteration 1901 / 8000) loss: 1.364826
(Time 9.52 sec; Iteration 1911 / 8000) loss: 1.266581
(Time 9.56 sec; Iteration 1921 / 8000) loss: 1.352975
```

```
(Time 9.60 sec; Iteration 1931 / 8000) loss: 1.417407
(Time 9.65 sec; Iteration 1941 / 8000) loss: 1.237848
(Time 9.69 sec; Iteration 1951 / 8000) loss: 1.352375
(Time 9.73 sec; Iteration 1961 / 8000) loss: 1.264502
(Time 9.78 sec; Iteration 1971 / 8000) loss: 1.476294
(Time 9.82 sec; Iteration 1981 / 8000) loss: 1.172755
(Time 9.86 sec; Iteration 1991 / 8000) loss: 1.470435
(Epoch 5 / 20) train acc: 0.517000; val_acc: 0.474700
(Time 10.01 sec; Iteration 2001 / 8000) loss: 1.412173
(Time 10.06 sec; Iteration 2011 / 8000) loss: 1.365631
(Time 10.10 sec; Iteration 2021 / 8000) loss: 1.380066
(Time 10.14 sec; Iteration 2031 / 8000) loss: 1.388442
(Time 10.18 sec; Iteration 2041 / 8000) loss: 1.406061
(Time 10.23 sec; Iteration 2051 / 8000) loss: 1.527548
(Time 10.27 sec; Iteration 2061 / 8000) loss: 1.423839
(Time 10.31 sec; Iteration 2071 / 8000) loss: 1.351541
(Time 10.36 sec; Iteration 2081 / 8000) loss: 1.392506
(Time 10.40 sec; Iteration 2091 / 8000) loss: 1.415820
(Time 10.44 sec; Iteration 2101 / 8000) loss: 1.368111
(Time 10.48 sec; Iteration 2111 / 8000) loss: 1.495955
(Time 10.53 sec; Iteration 2121 / 8000) loss: 1.439826
(Time 10.57 sec; Iteration 2131 / 8000) loss: 1.424299
(Time 10.61 sec; Iteration 2141 / 8000) loss: 1.480144
(Time 10.65 sec; Iteration 2151 / 8000) loss: 1.535295
(Time 10.70 sec; Iteration 2161 / 8000) loss: 1.348136
(Time 10.74 sec; Iteration 2171 / 8000) loss: 1.192357
(Time 10.78 sec; Iteration 2181 / 8000) loss: 1.474535
(Time 10.82 sec; Iteration 2191 / 8000) loss: 1.315101
(Time 10.87 sec; Iteration 2201 / 8000) loss: 1.386365
(Time 10.91 sec; Iteration 2211 / 8000) loss: 1.292390
(Time 10.95 sec; Iteration 2221 / 8000) loss: 1.374970
(Time 11.00 sec; Iteration 2231 / 8000) loss: 1.224959
(Time 11.04 sec; Iteration 2241 / 8000) loss: 1.395312
(Time 11.08 sec; Iteration 2251 / 8000) loss: 1.419005
(Time 11.13 sec; Iteration 2261 / 8000) loss: 1.312549
(Time 11.17 sec; Iteration 2271 / 8000) loss: 1.373220
(Time 11.21 sec; Iteration 2281 / 8000) loss: 1.416466
(Time 11.25 sec; Iteration 2291 / 8000) loss: 1.237260
(Time 11.30 sec; Iteration 2301 / 8000) loss: 1.500228
(Time 11.34 sec; Iteration 2311 / 8000) loss: 1.371111
(Time 11.38 sec; Iteration 2321 / 8000) loss: 1.398013
(Time 11.42 sec; Iteration 2331 / 8000) loss: 1.454128
(Time 11.47 sec; Iteration 2341 / 8000) loss: 1.293662
(Time 11.51 sec; Iteration 2351 / 8000) loss: 1.318695
(Time 11.55 sec; Iteration 2361 / 8000) loss: 1.328741
(Time 11.60 sec; Iteration 2371 / 8000) loss: 1.452432
(Time 11.64 sec; Iteration 2381 / 8000) loss: 1.607329
(Time 11.68 sec; Iteration 2391 / 8000) loss: 1.353413
```

```
(Epoch 6 / 20) train acc: 0.549000; val_acc: 0.486500
(Time 11.83 sec; Iteration 2401 / 8000) loss: 1.341348
(Time 11.88 sec; Iteration 2411 / 8000) loss: 1.454511
(Time 11.92 sec; Iteration 2421 / 8000) loss: 1.530261
(Time 11.96 sec; Iteration 2431 / 8000) loss: 1.269506
(Time 12.00 sec; Iteration 2441 / 8000) loss: 1.215866
(Time 12.05 sec; Iteration 2451 / 8000) loss: 1.234394
(Time 12.09 sec; Iteration 2461 / 8000) loss: 1.497389
(Time 12.13 sec; Iteration 2471 / 8000) loss: 1.437315
(Time 12.18 sec; Iteration 2481 / 8000) loss: 1.112067
(Time 12.22 sec; Iteration 2491 / 8000) loss: 1.571989
(Time 12.26 sec; Iteration 2501 / 8000) loss: 1.347804
(Time 12.30 sec; Iteration 2511 / 8000) loss: 1.473773
(Time 12.34 sec; Iteration 2521 / 8000) loss: 1.311973
(Time 12.39 sec; Iteration 2531 / 8000) loss: 1.300632
(Time 12.43 sec; Iteration 2541 / 8000) loss: 1.323950
(Time 12.47 sec; Iteration 2551 / 8000) loss: 1.197390
(Time 12.52 sec; Iteration 2561 / 8000) loss: 1.297808
(Time 12.56 sec; Iteration 2571 / 8000) loss: 1.443908
(Time 12.60 sec; Iteration 2581 / 8000) loss: 1.362915
(Time 12.64 sec; Iteration 2591 / 8000) loss: 1.387469
(Time 12.68 sec; Iteration 2601 / 8000) loss: 1.197494
(Time 12.73 sec; Iteration 2611 / 8000) loss: 1.331915
(Time 12.77 sec; Iteration 2621 / 8000) loss: 1.407377
(Time 12.81 sec; Iteration 2631 / 8000) loss: 1.348649
(Time 12.86 sec; Iteration 2641 / 8000) loss: 1.522015
(Time 12.90 sec; Iteration 2651 / 8000) loss: 1.474020
(Time 12.95 sec; Iteration 2661 / 8000) loss: 1.439386
(Time 12.99 sec; Iteration 2671 / 8000) loss: 1.443060
(Time 13.03 sec; Iteration 2681 / 8000) loss: 1.181620
(Time 13.07 sec; Iteration 2691 / 8000) loss: 1.268568
(Time 13.12 sec; Iteration 2701 / 8000) loss: 1.278015
(Time 13.16 sec; Iteration 2711 / 8000) loss: 1.255225
(Time 13.20 sec; Iteration 2721 / 8000) loss: 1.162669
(Time 13.24 sec; Iteration 2731 / 8000) loss: 1.227514
(Time 13.29 sec; Iteration 2741 / 8000) loss: 1.356799
(Time 13.33 sec; Iteration 2751 / 8000) loss: 1.235003
(Time 13.37 sec; Iteration 2761 / 8000) loss: 1.399809
(Time 13.42 sec; Iteration 2771 / 8000) loss: 1.254974
(Time 13.46 sec; Iteration 2781 / 8000) loss: 1.417478
(Time 13.50 sec; Iteration 2791 / 8000) loss: 1.201602
(Epoch 7 / 20) train acc: 0.539000; val_acc: 0.489900
(Time 13.65 sec; Iteration 2801 / 8000) loss: 1.325908
(Time 13.69 sec; Iteration 2811 / 8000) loss: 1.259086
(Time 13.73 sec; Iteration 2821 / 8000) loss: 1.363871
(Time 13.78 sec; Iteration 2831 / 8000) loss: 1.341692
(Time 13.82 sec; Iteration 2841 / 8000) loss: 1.180890
(Time 13.86 sec; Iteration 2851 / 8000) loss: 1.124726
```

```
(Time 13.90 sec; Iteration 2861 / 8000) loss: 1.398927
(Time 13.95 sec; Iteration 2871 / 8000) loss: 1.309478
(Time 13.99 sec; Iteration 2881 / 8000) loss: 1.401368
(Time 14.04 sec; Iteration 2891 / 8000) loss: 1.377041
(Time 14.08 sec; Iteration 2901 / 8000) loss: 1.318557
(Time 14.12 sec; Iteration 2911 / 8000) loss: 1.292226
(Time 14.17 sec; Iteration 2921 / 8000) loss: 1.508426
(Time 14.21 sec; Iteration 2931 / 8000) loss: 1.409223
(Time 14.25 sec; Iteration 2941 / 8000) loss: 1.149919
(Time 14.30 sec; Iteration 2951 / 8000) loss: 1.287884
(Time 14.34 sec; Iteration 2961 / 8000) loss: 1.353141
(Time 14.38 sec; Iteration 2971 / 8000) loss: 1.368842
(Time 14.42 sec; Iteration 2981 / 8000) loss: 1.188245
(Time 14.47 sec; Iteration 2991 / 8000) loss: 1.343926
(Time 14.51 sec; Iteration 3001 / 8000) loss: 1.470120
(Time 14.55 sec; Iteration 3011 / 8000) loss: 1.311723
(Time 14.59 sec; Iteration 3021 / 8000) loss: 1.390933
(Time 14.64 sec; Iteration 3031 / 8000) loss: 1.181671
(Time 14.68 sec; Iteration 3041 / 8000) loss: 1.360569
(Time 14.72 sec; Iteration 3051 / 8000) loss: 1.254770
(Time 14.77 sec; Iteration 3061 / 8000) loss: 1.226239
(Time 14.81 sec; Iteration 3071 / 8000) loss: 1.171366
(Time 14.85 sec; Iteration 3081 / 8000) loss: 1.206862
(Time 14.89 sec; Iteration 3091 / 8000) loss: 1.231305
(Time 14.94 sec; Iteration 3101 / 8000) loss: 1.246286
(Time 14.98 sec; Iteration 3111 / 8000) loss: 1.442142
(Time 15.03 sec; Iteration 3121 / 8000) loss: 1.204228
(Time 15.07 sec; Iteration 3131 / 8000) loss: 1.289332
(Time 15.12 sec; Iteration 3141 / 8000) loss: 1.269922
(Time 15.17 sec; Iteration 3151 / 8000) loss: 1.245389
(Time 15.22 sec; Iteration 3161 / 8000) loss: 1.221220
(Time 15.27 sec; Iteration 3171 / 8000) loss: 1.161363
(Time 15.32 sec; Iteration 3181 / 8000) loss: 1.381036
(Time 15.37 sec; Iteration 3191 / 8000) loss: 1.286729
(Epoch 8 / 20) train acc: 0.529000; val_acc: 0.496400
(Time 15.52 sec; Iteration 3201 / 8000) loss: 1.180377
(Time 15.57 sec; Iteration 3211 / 8000) loss: 1.409708
(Time 15.62 sec; Iteration 3221 / 8000) loss: 1.242363
(Time 15.67 sec; Iteration 3231 / 8000) loss: 1.109249
(Time 15.72 sec; Iteration 3241 / 8000) loss: 1.294702
(Time 15.77 sec; Iteration 3251 / 8000) loss: 1.317802
(Time 15.81 sec; Iteration 3261 / 8000) loss: 1.347546
(Time 15.86 sec; Iteration 3271 / 8000) loss: 1.235995
(Time 15.91 sec; Iteration 3281 / 8000) loss: 1.380530
(Time 15.95 sec; Iteration 3291 / 8000) loss: 1.381805
(Time 16.01 sec; Iteration 3301 / 8000) loss: 1.360922
(Time 16.06 sec; Iteration 3311 / 8000) loss: 1.268379
(Time 16.10 sec; Iteration 3321 / 8000) loss: 1.218190
```

```
(Time 16.15 sec; Iteration 3331 / 8000) loss: 1.279020
(Time 16.20 sec; Iteration 3341 / 8000) loss: 1.219954
(Time 16.24 sec; Iteration 3351 / 8000) loss: 1.212037
(Time 16.29 sec; Iteration 3361 / 8000) loss: 1.360056
(Time 16.34 sec; Iteration 3371 / 8000) loss: 1.275914
(Time 16.39 sec; Iteration 3381 / 8000) loss: 1.101094
(Time 16.44 sec; Iteration 3391 / 8000) loss: 1.436274
(Time 16.49 sec; Iteration 3401 / 8000) loss: 1.390154
(Time 16.53 sec; Iteration 3411 / 8000) loss: 1.142683
(Time 16.58 sec; Iteration 3421 / 8000) loss: 1.312863
(Time 16.63 sec; Iteration 3431 / 8000) loss: 1.263322
(Time 16.68 sec; Iteration 3441 / 8000) loss: 1.453143
(Time 16.73 sec; Iteration 3451 / 8000) loss: 1.384671
(Time 16.78 sec; Iteration 3461 / 8000) loss: 1.509826
(Time 16.83 sec; Iteration 3471 / 8000) loss: 1.198599
(Time 16.88 sec; Iteration 3481 / 8000) loss: 1.238305
(Time 16.93 sec; Iteration 3491 / 8000) loss: 1.242721
(Time 16.98 sec; Iteration 3501 / 8000) loss: 1.126090
(Time 17.03 sec; Iteration 3511 / 8000) loss: 1.109522
(Time 17.08 sec; Iteration 3521 / 8000) loss: 1.393798
(Time 17.13 sec; Iteration 3531 / 8000) loss: 1.260682
(Time 17.17 sec; Iteration 3541 / 8000) loss: 1.296079
(Time 17.23 sec; Iteration 3551 / 8000) loss: 1.252848
(Time 17.28 sec; Iteration 3561 / 8000) loss: 1.365798
(Time 17.33 sec; Iteration 3571 / 8000) loss: 1.337792
(Time 17.38 sec; Iteration 3581 / 8000) loss: 1.396553
(Time 17.43 sec; Iteration 3591 / 8000) loss: 1.414233
(Epoch 9 / 20) train acc: 0.568000; val_acc: 0.506900
(Time 17.59 sec; Iteration 3601 / 8000) loss: 1.471752
(Time 17.65 sec; Iteration 3611 / 8000) loss: 1.306402
(Time 17.70 sec; Iteration 3621 / 8000) loss: 1.383040
(Time 17.75 sec; Iteration 3631 / 8000) loss: 1.384934
(Time 17.80 sec; Iteration 3641 / 8000) loss: 1.227346
(Time 17.85 sec; Iteration 3651 / 8000) loss: 1.154861
(Time 17.90 sec; Iteration 3661 / 8000) loss: 1.203838
(Time 17.95 sec; Iteration 3671 / 8000) loss: 1.268199
(Time 18.00 sec; Iteration 3681 / 8000) loss: 1.264610
(Time 18.05 sec; Iteration 3691 / 8000) loss: 1.090994
(Time 18.10 sec; Iteration 3701 / 8000) loss: 1.214736
(Time 18.15 sec; Iteration 3711 / 8000) loss: 1.398486
(Time 18.20 sec; Iteration 3721 / 8000) loss: 1.376327
(Time 18.25 sec; Iteration 3731 / 8000) loss: 1.272982
(Time 18.30 sec; Iteration 3741 / 8000) loss: 1.278452
(Time 18.35 sec; Iteration 3751 / 8000) loss: 1.165819
(Time 18.41 sec; Iteration 3761 / 8000) loss: 1.259966
(Time 18.46 sec; Iteration 3771 / 8000) loss: 1.371037
(Time 18.51 sec; Iteration 3781 / 8000) loss: 1.139055
(Time 18.56 sec; Iteration 3791 / 8000) loss: 1.253244
```

```
(Time 18.61 sec; Iteration 3801 / 8000) loss: 1.424030
(Time 18.65 sec; Iteration 3811 / 8000) loss: 1.281586
(Time 18.69 sec; Iteration 3821 / 8000) loss: 1.373167
(Time 18.73 sec; Iteration 3831 / 8000) loss: 1.254153
(Time 18.78 sec; Iteration 3841 / 8000) loss: 1.246466
(Time 18.82 sec; Iteration 3851 / 8000) loss: 1.205352
(Time 18.86 sec; Iteration 3861 / 8000) loss: 1.563120
(Time 18.91 sec; Iteration 3871 / 8000) loss: 1.237035
(Time 18.95 sec; Iteration 3881 / 8000) loss: 1.333726
(Time 18.99 sec; Iteration 3891 / 8000) loss: 1.190968
(Time 19.03 sec; Iteration 3901 / 8000) loss: 1.442500
(Time 19.08 sec; Iteration 3911 / 8000) loss: 1.306660
(Time 19.12 sec; Iteration 3921 / 8000) loss: 1.037206
(Time 19.16 sec; Iteration 3931 / 8000) loss: 1.385250
(Time 19.21 sec; Iteration 3941 / 8000) loss: 1.100189
(Time 19.25 sec; Iteration 3951 / 8000) loss: 1.295998
(Time 19.29 sec; Iteration 3961 / 8000) loss: 1.167010
(Time 19.34 sec; Iteration 3971 / 8000) loss: 1.510381
(Time 19.38 sec; Iteration 3981 / 8000) loss: 1.384203
(Time 19.43 sec; Iteration 3991 / 8000) loss: 1.292444
(Epoch 10 / 20) train acc: 0.581000; val_acc: 0.506100
(Time 19.58 sec; Iteration 4001 / 8000) loss: 1.260839
(Time 19.62 sec; Iteration 4011 / 8000) loss: 1.413273
(Time 19.66 sec; Iteration 4021 / 8000) loss: 1.337102
(Time 19.70 sec; Iteration 4031 / 8000) loss: 1.337654
(Time 19.74 sec; Iteration 4041 / 8000) loss: 1.264559
(Time 19.79 sec; Iteration 4051 / 8000) loss: 1.317628
(Time 19.83 sec; Iteration 4061 / 8000) loss: 1.186777
(Time 19.88 sec; Iteration 4071 / 8000) loss: 1.176233
(Time 19.92 sec; Iteration 4081 / 8000) loss: 1.219115
(Time 19.96 sec; Iteration 4091 / 8000) loss: 1.312596
(Time 20.00 sec; Iteration 4101 / 8000) loss: 1.343185
(Time 20.05 sec; Iteration 4111 / 8000) loss: 1.152542
(Time 20.09 sec; Iteration 4121 / 8000) loss: 1.299046
(Time 20.13 sec; Iteration 4131 / 8000) loss: 1.173922
(Time 20.18 sec; Iteration 4141 / 8000) loss: 1.147126
(Time 20.22 sec; Iteration 4151 / 8000) loss: 1.278966
(Time 20.26 sec; Iteration 4161 / 8000) loss: 1.149381
(Time 20.30 sec; Iteration 4171 / 8000) loss: 1.105889
(Time 20.35 sec; Iteration 4181 / 8000) loss: 1.243304
(Time 20.39 sec; Iteration 4191 / 8000) loss: 1.266655
(Time 20.43 sec; Iteration 4201 / 8000) loss: 1.227603
(Time 20.47 sec; Iteration 4211 / 8000) loss: 1.129975
(Time 20.52 sec; Iteration 4221 / 8000) loss: 1.188917
(Time 20.56 sec; Iteration 4231 / 8000) loss: 1.203324
(Time 20.60 sec; Iteration 4241 / 8000) loss: 1.205165
(Time 20.64 sec; Iteration 4251 / 8000) loss: 1.291823
(Time 20.69 sec; Iteration 4261 / 8000) loss: 1.357063
```

```
(Time 20.73 sec; Iteration 4271 / 8000) loss: 1.121283
(Time 20.77 sec; Iteration 4281 / 8000) loss: 1.176875
(Time 20.81 sec; Iteration 4291 / 8000) loss: 1.213360
(Time 20.86 sec; Iteration 4301 / 8000) loss: 1.179129
(Time 20.90 sec; Iteration 4311 / 8000) loss: 1.427982
(Time 20.94 sec; Iteration 4321 / 8000) loss: 1.092745
(Time 20.98 sec; Iteration 4331 / 8000) loss: 1.116980
(Time 21.03 sec; Iteration 4341 / 8000) loss: 1.146390
(Time 21.07 sec; Iteration 4351 / 8000) loss: 1.178112
(Time 21.11 sec; Iteration 4361 / 8000) loss: 1.170790
(Time 21.16 sec; Iteration 4371 / 8000) loss: 1.276004
(Time 21.20 sec; Iteration 4381 / 8000) loss: 1.017676
(Time 21.25 sec; Iteration 4391 / 8000) loss: 1.114798
(Epoch 11 / 20) train acc: 0.614000; val_acc: 0.513600
(Time 21.40 sec; Iteration 4401 / 8000) loss: 1.245968
(Time 21.44 sec; Iteration 4411 / 8000) loss: 1.232027
(Time 21.49 sec; Iteration 4421 / 8000) loss: 1.181578
(Time 21.53 sec; Iteration 4431 / 8000) loss: 1.030650
(Time 21.58 sec; Iteration 4441 / 8000) loss: 1.215218
(Time 21.62 sec; Iteration 4451 / 8000) loss: 1.120076
(Time 21.66 sec; Iteration 4461 / 8000) loss: 1.387417
(Time 21.70 sec; Iteration 4471 / 8000) loss: 1.309303
(Time 21.75 sec; Iteration 4481 / 8000) loss: 1.166170
(Time 21.79 sec; Iteration 4491 / 8000) loss: 1.187347
(Time 21.83 sec; Iteration 4501 / 8000) loss: 1.266367
(Time 21.88 sec; Iteration 4511 / 8000) loss: 1.303981
(Time 21.92 sec; Iteration 4521 / 8000) loss: 1.304114
(Time 21.96 sec; Iteration 4531 / 8000) loss: 1.226509
(Time 22.00 sec; Iteration 4541 / 8000) loss: 1.154562
(Time 22.05 sec; Iteration 4551 / 8000) loss: 1.269596
(Time 22.09 sec; Iteration 4561 / 8000) loss: 1.206417
(Time 22.13 sec; Iteration 4571 / 8000) loss: 1.096264
(Time 22.18 sec; Iteration 4581 / 8000) loss: 1.219308
(Time 22.22 sec; Iteration 4591 / 8000) loss: 1.353126
(Time 22.27 sec; Iteration 4601 / 8000) loss: 1.180343
(Time 22.31 sec; Iteration 4611 / 8000) loss: 1.113593
(Time 22.35 sec; Iteration 4621 / 8000) loss: 1.114961
(Time 22.39 sec; Iteration 4631 / 8000) loss: 1.201712
(Time 22.44 sec; Iteration 4641 / 8000) loss: 1.184255
(Time 22.48 sec; Iteration 4651 / 8000) loss: 1.199808
(Time 22.52 sec; Iteration 4661 / 8000) loss: 1.122299
(Time 22.56 sec; Iteration 4671 / 8000) loss: 1.129043
(Time 22.61 sec; Iteration 4681 / 8000) loss: 1.191901
(Time 22.65 sec; Iteration 4691 / 8000) loss: 1.311126
(Time 22.69 sec; Iteration 4701 / 8000) loss: 1.207893
(Time 22.74 sec; Iteration 4711 / 8000) loss: 1.302956
(Time 22.78 sec; Iteration 4721 / 8000) loss: 1.176415
(Time 22.82 sec; Iteration 4731 / 8000) loss: 1.245183
```

```
(Time 22.87 sec; Iteration 4741 / 8000) loss: 1.064497
(Time 22.91 sec; Iteration 4751 / 8000) loss: 1.232743
(Time 22.96 sec; Iteration 4761 / 8000) loss: 1.260703
(Time 23.00 sec; Iteration 4771 / 8000) loss: 1.179166
(Time 23.04 sec; Iteration 4781 / 8000) loss: 1.340145
(Time 23.08 sec; Iteration 4791 / 8000) loss: 1.148120
(Epoch 12 / 20) train acc: 0.575000; val_acc: 0.511900
(Time 23.24 sec; Iteration 4801 / 8000) loss: 1.184140
(Time 23.28 sec; Iteration 4811 / 8000) loss: 1.297443
(Time 23.32 sec; Iteration 4821 / 8000) loss: 1.097664
(Time 23.37 sec; Iteration 4831 / 8000) loss: 1.085454
(Time 23.41 sec; Iteration 4841 / 8000) loss: 1.179471
(Time 23.45 sec; Iteration 4851 / 8000) loss: 1.121438
(Time 23.49 sec; Iteration 4861 / 8000) loss: 1.202191
(Time 23.53 sec; Iteration 4871 / 8000) loss: 1.096466
(Time 23.58 sec; Iteration 4881 / 8000) loss: 1.262228
(Time 23.62 sec; Iteration 4891 / 8000) loss: 1.127605
(Time 23.66 sec; Iteration 4901 / 8000) loss: 1.172423
(Time 23.70 sec; Iteration 4911 / 8000) loss: 1.245361
(Time 23.75 sec; Iteration 4921 / 8000) loss: 1.069427
(Time 23.79 sec; Iteration 4931 / 8000) loss: 1.079448
(Time 23.83 sec; Iteration 4941 / 8000) loss: 1.199983
(Time 23.87 sec; Iteration 4951 / 8000) loss: 1.211033
(Time 23.92 sec; Iteration 4961 / 8000) loss: 1.218202
(Time 23.96 sec; Iteration 4971 / 8000) loss: 1.335323
(Time 24.00 sec; Iteration 4981 / 8000) loss: 1.234009
(Time 24.04 sec; Iteration 4991 / 8000) loss: 1.166646
(Time 24.09 sec; Iteration 5001 / 8000) loss: 1.115607
(Time 24.13 sec; Iteration 5011 / 8000) loss: 1.171114
(Time 24.18 sec; Iteration 5021 / 8000) loss: 1.153437
(Time 24.22 sec; Iteration 5031 / 8000) loss: 1.262027
(Time 24.26 sec; Iteration 5041 / 8000) loss: 1.235849
(Time 24.31 sec; Iteration 5051 / 8000) loss: 0.979780
(Time 24.35 sec; Iteration 5061 / 8000) loss: 1.318055
(Time 24.39 sec; Iteration 5071 / 8000) loss: 1.157569
(Time 24.43 sec; Iteration 5081 / 8000) loss: 1.152742
(Time 24.48 sec; Iteration 5091 / 8000) loss: 1.190425
(Time 24.52 sec; Iteration 5101 / 8000) loss: 1.148353
(Time 24.56 sec; Iteration 5111 / 8000) loss: 1.342662
(Time 24.61 sec; Iteration 5121 / 8000) loss: 1.199204
(Time 24.65 sec; Iteration 5131 / 8000) loss: 1.097798
(Time 24.70 sec; Iteration 5141 / 8000) loss: 1.185935
(Time 24.76 sec; Iteration 5151 / 8000) loss: 1.045423
(Time 24.89 sec; Iteration 5161 / 8000) loss: 1.139327
(Time 25.01 sec; Iteration 5171 / 8000) loss: 1.237807
(Time 25.07 sec; Iteration 5181 / 8000) loss: 1.261219
(Time 25.12 sec; Iteration 5191 / 8000) loss: 1.176166
(Epoch 13 / 20) train acc: 0.584000; val_acc: 0.514100
```

```
(Time 25.28 sec; Iteration 5201 / 8000) loss: 1.218754
(Time 25.32 sec; Iteration 5211 / 8000) loss: 0.949391
(Time 25.37 sec; Iteration 5221 / 8000) loss: 1.120123
(Time 25.41 sec; Iteration 5231 / 8000) loss: 1.209788
(Time 25.45 sec; Iteration 5241 / 8000) loss: 1.204999
(Time 25.50 sec; Iteration 5251 / 8000) loss: 1.205357
(Time 25.54 sec; Iteration 5261 / 8000) loss: 1.135834
(Time 25.58 sec; Iteration 5271 / 8000) loss: 1.198687
(Time 25.65 sec; Iteration 5281 / 8000) loss: 1.045356
(Time 25.77 sec; Iteration 5291 / 8000) loss: 1.156109
(Time 25.90 sec; Iteration 5301 / 8000) loss: 1.238660
(Time 25.95 sec; Iteration 5311 / 8000) loss: 1.066728
(Time 26.03 sec; Iteration 5321 / 8000) loss: 1.240030
(Time 26.08 sec; Iteration 5331 / 8000) loss: 1.103401
(Time 26.12 sec; Iteration 5341 / 8000) loss: 1.219692
(Time 26.16 sec; Iteration 5351 / 8000) loss: 1.090398
(Time 26.21 sec; Iteration 5361 / 8000) loss: 1.213761
(Time 26.25 sec; Iteration 5371 / 8000) loss: 1.000552
(Time 26.30 sec; Iteration 5381 / 8000) loss: 1.311897
(Time 26.34 sec; Iteration 5391 / 8000) loss: 1.118381
(Time 26.38 sec; Iteration 5401 / 8000) loss: 1.312198
(Time 26.43 sec; Iteration 5411 / 8000) loss: 1.121567
(Time 26.47 sec; Iteration 5421 / 8000) loss: 1.195857
(Time 26.51 sec; Iteration 5431 / 8000) loss: 1.206903
(Time 26.55 sec; Iteration 5441 / 8000) loss: 1.150481
(Time 26.62 sec; Iteration 5451 / 8000) loss: 1.108264
(Time 26.76 sec; Iteration 5461 / 8000) loss: 1.270943
(Time 26.88 sec; Iteration 5471 / 8000) loss: 1.103286
(Time 26.96 sec; Iteration 5481 / 8000) loss: 1.225827
(Time 27.01 sec; Iteration 5491 / 8000) loss: 1.031693
(Time 27.05 sec; Iteration 5501 / 8000) loss: 1.019815
(Time 27.09 sec; Iteration 5511 / 8000) loss: 1.187512
(Time 27.13 sec; Iteration 5521 / 8000) loss: 1.129688
(Time 27.18 sec; Iteration 5531 / 8000) loss: 1.122110
(Time 27.22 sec; Iteration 5541 / 8000) loss: 1.027762
(Time 27.26 sec; Iteration 5551 / 8000) loss: 1.179056
(Time 27.31 sec; Iteration 5561 / 8000) loss: 1.176161
(Time 27.35 sec; Iteration 5571 / 8000) loss: 1.226215
(Time 27.39 sec; Iteration 5581 / 8000) loss: 1.207482
(Time 27.44 sec; Iteration 5591 / 8000) loss: 1.186410
(Epoch 14 / 20) train acc: 0.579000; val_acc: 0.518300
(Time 27.59 sec; Iteration 5601 / 8000) loss: 1.176066
(Time 27.63 sec; Iteration 5611 / 8000) loss: 1.075066
(Time 27.68 sec; Iteration 5621 / 8000) loss: 1.092907
(Time 27.72 sec; Iteration 5631 / 8000) loss: 1.117329
(Time 27.76 sec; Iteration 5641 / 8000) loss: 1.146648
(Time 27.80 sec; Iteration 5651 / 8000) loss: 1.152727
(Time 27.85 sec; Iteration 5661 / 8000) loss: 0.879547
```

```
(Time 27.89 sec; Iteration 5671 / 8000) loss: 0.989005
(Time 27.93 sec; Iteration 5681 / 8000) loss: 1.243404
(Time 27.97 sec; Iteration 5691 / 8000) loss: 1.000434
(Time 28.01 sec; Iteration 5701 / 8000) loss: 1.033780
(Time 28.06 sec; Iteration 5711 / 8000) loss: 1.134701
(Time 28.10 sec; Iteration 5721 / 8000) loss: 1.095631
(Time 28.15 sec; Iteration 5731 / 8000) loss: 1.156347
(Time 28.19 sec; Iteration 5741 / 8000) loss: 1.157606
(Time 28.23 sec; Iteration 5751 / 8000) loss: 1.167874
(Time 28.27 sec; Iteration 5761 / 8000) loss: 1.162497
(Time 28.31 sec; Iteration 5771 / 8000) loss: 1.187374
(Time 28.36 sec; Iteration 5781 / 8000) loss: 1.033929
(Time 28.40 sec; Iteration 5791 / 8000) loss: 1.016022
(Time 28.45 sec; Iteration 5801 / 8000) loss: 1.193802
(Time 28.49 sec; Iteration 5811 / 8000) loss: 1.177333
(Time 28.53 sec; Iteration 5821 / 8000) loss: 0.965609
(Time 28.58 sec; Iteration 5831 / 8000) loss: 1.370536
(Time 28.63 sec; Iteration 5841 / 8000) loss: 1.257248
(Time 28.68 sec; Iteration 5851 / 8000) loss: 1.160465
(Time 28.73 sec; Iteration 5861 / 8000) loss: 1.045980
(Time 28.78 sec; Iteration 5871 / 8000) loss: 1.004162
(Time 28.83 sec; Iteration 5881 / 8000) loss: 1.171955
(Time 28.88 sec; Iteration 5891 / 8000) loss: 1.252793
(Time 28.93 sec; Iteration 5901 / 8000) loss: 1.311589
(Time 28.97 sec; Iteration 5911 / 8000) loss: 1.041840
(Time 29.03 sec; Iteration 5921 / 8000) loss: 1.184844
(Time 29.08 sec; Iteration 5931 / 8000) loss: 1.256646
(Time 29.12 sec; Iteration 5941 / 8000) loss: 1.181448
(Time 29.17 sec; Iteration 5951 / 8000) loss: 1.182284
(Time 29.22 sec; Iteration 5961 / 8000) loss: 1.255372
(Time 29.28 sec; Iteration 5971 / 8000) loss: 1.037218
(Time 29.32 sec; Iteration 5981 / 8000) loss: 1.161977
(Time 29.37 sec; Iteration 5991 / 8000) loss: 1.098796
(Epoch 15 / 20) train acc: 0.622000; val_acc: 0.516500
(Time 29.53 sec; Iteration 6001 / 8000) loss: 1.078514
(Time 29.58 sec; Iteration 6011 / 8000) loss: 1.158509
(Time 29.62 sec; Iteration 6021 / 8000) loss: 1.306555
(Time 29.67 sec; Iteration 6031 / 8000) loss: 1.094071
(Time 29.72 sec; Iteration 6041 / 8000) loss: 1.136284
(Time 29.77 sec; Iteration 6051 / 8000) loss: 1.226913
(Time 29.82 sec; Iteration 6061 / 8000) loss: 1.312909
(Time 29.86 sec; Iteration 6071 / 8000) loss: 1.004155
(Time 29.91 sec; Iteration 6081 / 8000) loss: 1.108671
(Time 29.96 sec; Iteration 6091 / 8000) loss: 1.120606
(Time 30.01 sec; Iteration 6101 / 8000) loss: 1.261905
(Time 30.06 sec; Iteration 6111 / 8000) loss: 1.190173
(Time 30.11 sec; Iteration 6121 / 8000) loss: 1.155047
(Time 30.16 sec; Iteration 6131 / 8000) loss: 1.269598
```

```
(Time 30.21 sec; Iteration 6141 / 8000) loss: 1.177689
(Time 30.27 sec; Iteration 6151 / 8000) loss: 1.035619
(Time 30.31 sec; Iteration 6161 / 8000) loss: 1.158220
(Time 30.36 sec; Iteration 6171 / 8000) loss: 1.211366
(Time 30.41 sec; Iteration 6181 / 8000) loss: 1.232188
(Time 30.46 sec; Iteration 6191 / 8000) loss: 1.178029
(Time 30.51 sec; Iteration 6201 / 8000) loss: 1.079571
(Time 30.56 sec; Iteration 6211 / 8000) loss: 1.223659
(Time 30.60 sec; Iteration 6221 / 8000) loss: 0.960388
(Time 30.65 sec; Iteration 6231 / 8000) loss: 1.264415
(Time 30.70 sec; Iteration 6241 / 8000) loss: 1.199141
(Time 30.75 sec; Iteration 6251 / 8000) loss: 1.177021
(Time 30.81 sec; Iteration 6261 / 8000) loss: 1.153305
(Time 30.85 sec; Iteration 6271 / 8000) loss: 0.996014
(Time 30.90 sec; Iteration 6281 / 8000) loss: 1.104129
(Time 30.95 sec; Iteration 6291 / 8000) loss: 1.165566
(Time 31.01 sec; Iteration 6301 / 8000) loss: 1.158667
(Time 31.06 sec; Iteration 6311 / 8000) loss: 1.121371
(Time 31.11 sec; Iteration 6321 / 8000) loss: 1.083281
(Time 31.15 sec; Iteration 6331 / 8000) loss: 1.046354
(Time 31.20 sec; Iteration 6341 / 8000) loss: 1.230958
(Time 31.25 sec; Iteration 6351 / 8000) loss: 1.088088
(Time 31.30 sec; Iteration 6361 / 8000) loss: 1.097740
(Time 31.35 sec; Iteration 6371 / 8000) loss: 1.148480
(Time 31.40 sec; Iteration 6381 / 8000) loss: 1.073526
(Time 31.45 sec; Iteration 6391 / 8000) loss: 1.216074
(Epoch 16 / 20) train acc: 0.560000; val_acc: 0.520900
(Time 31.61 sec; Iteration 6401 / 8000) loss: 1.252698
(Time 31.66 sec; Iteration 6411 / 8000) loss: 1.198587
(Time 31.71 sec; Iteration 6421 / 8000) loss: 1.100727
(Time 31.76 sec; Iteration 6431 / 8000) loss: 1.133145
(Time 31.81 sec; Iteration 6441 / 8000) loss: 1.255859
(Time 31.86 sec; Iteration 6451 / 8000) loss: 1.012849
(Time 31.91 sec; Iteration 6461 / 8000) loss: 1.216098
(Time 31.96 sec; Iteration 6471 / 8000) loss: 1.122756
(Time 32.01 sec; Iteration 6481 / 8000) loss: 0.926522
(Time 32.05 sec; Iteration 6491 / 8000) loss: 1.102520
(Time 32.10 sec; Iteration 6501 / 8000) loss: 0.913332
(Time 32.14 sec; Iteration 6511 / 8000) loss: 1.332486
(Time 32.18 sec; Iteration 6521 / 8000) loss: 1.121745
(Time 32.22 sec; Iteration 6531 / 8000) loss: 1.013009
(Time 32.27 sec; Iteration 6541 / 8000) loss: 1.141801
(Time 32.31 sec; Iteration 6551 / 8000) loss: 1.153953
(Time 32.35 sec; Iteration 6561 / 8000) loss: 1.113425
(Time 32.40 sec; Iteration 6571 / 8000) loss: 1.157148
(Time 32.44 sec; Iteration 6581 / 8000) loss: 1.141855
(Time 32.48 sec; Iteration 6591 / 8000) loss: 1.156055
(Time 32.53 sec; Iteration 6601 / 8000) loss: 1.018054
```

```
(Time 32.58 sec; Iteration 6611 / 8000) loss: 1.216052
(Time 32.62 sec; Iteration 6621 / 8000) loss: 1.280552
(Time 32.67 sec; Iteration 6631 / 8000) loss: 1.318749
(Time 32.71 sec; Iteration 6641 / 8000) loss: 1.130039
(Time 32.75 sec; Iteration 6651 / 8000) loss: 1.068184
(Time 32.80 sec; Iteration 6661 / 8000) loss: 1.002914
(Time 32.84 sec; Iteration 6671 / 8000) loss: 1.145888
(Time 32.88 sec; Iteration 6681 / 8000) loss: 1.240326
(Time 32.93 sec; Iteration 6691 / 8000) loss: 1.094868
(Time 32.97 sec; Iteration 6701 / 8000) loss: 1.107030
(Time 33.01 sec; Iteration 6711 / 8000) loss: 1.103841
(Time 33.06 sec; Iteration 6721 / 8000) loss: 1.276179
(Time 33.10 sec; Iteration 6731 / 8000) loss: 1.266849
(Time 33.14 sec; Iteration 6741 / 8000) loss: 1.220176
(Time 33.19 sec; Iteration 6751 / 8000) loss: 1.282358
(Time 33.23 sec; Iteration 6761 / 8000) loss: 1.009467
(Time 33.27 sec; Iteration 6771 / 8000) loss: 1.154689
(Time 33.31 sec; Iteration 6781 / 8000) loss: 1.241300
(Time 33.36 sec; Iteration 6791 / 8000) loss: 1.256572
(Epoch 17 / 20) train acc: 0.620000; val_acc: 0.519500
(Time 33.51 sec; Iteration 6801 / 8000) loss: 1.151431
(Time 33.55 sec; Iteration 6811 / 8000) loss: 1.218204
(Time 33.59 sec; Iteration 6821 / 8000) loss: 1.076329
(Time 33.64 sec; Iteration 6831 / 8000) loss: 1.044177
(Time 33.68 sec; Iteration 6841 / 8000) loss: 1.261696
(Time 33.72 sec; Iteration 6851 / 8000) loss: 1.020929
(Time 33.77 sec; Iteration 6861 / 8000) loss: 1.030695
(Time 33.81 sec; Iteration 6871 / 8000) loss: 1.088330
(Time 33.85 sec; Iteration 6881 / 8000) loss: 1.261959
(Time 33.89 sec; Iteration 6891 / 8000) loss: 1.138552
(Time 33.94 sec; Iteration 6901 / 8000) loss: 1.130234
(Time 33.98 sec; Iteration 6911 / 8000) loss: 1.170873
(Time 34.02 sec; Iteration 6921 / 8000) loss: 1.031831
(Time 34.06 sec; Iteration 6931 / 8000) loss: 1.111410
(Time 34.11 sec; Iteration 6941 / 8000) loss: 1.019988
(Time 34.15 sec; Iteration 6951 / 8000) loss: 1.175702
(Time 34.19 sec; Iteration 6961 / 8000) loss: 1.158080
(Time 34.24 sec; Iteration 6971 / 8000) loss: 1.061829
(Time 34.28 sec; Iteration 6981 / 8000) loss: 1.149335
(Time 34.32 sec; Iteration 6991 / 8000) loss: 1.104751
(Time 34.36 sec; Iteration 7001 / 8000) loss: 1.050101
(Time 34.41 sec; Iteration 7011 / 8000) loss: 1.158775
(Time 34.45 sec; Iteration 7021 / 8000) loss: 1.100383
(Time 34.49 sec; Iteration 7031 / 8000) loss: 1.041710
(Time 34.54 sec; Iteration 7041 / 8000) loss: 1.111438
(Time 34.59 sec; Iteration 7051 / 8000) loss: 0.938747
(Time 34.64 sec; Iteration 7061 / 8000) loss: 1.242225
(Time 34.68 sec; Iteration 7071 / 8000) loss: 0.981588
```

```
(Time 34.72 sec; Iteration 7081 / 8000) loss: 1.138213
(Time 34.76 sec; Iteration 7091 / 8000) loss: 1.090152
(Time 34.81 sec; Iteration 7101 / 8000) loss: 1.115758
(Time 34.85 sec; Iteration 7111 / 8000) loss: 0.954086
(Time 34.89 sec; Iteration 7121 / 8000) loss: 1.105737
(Time 34.94 sec; Iteration 7131 / 8000) loss: 1.260969
(Time 34.98 sec; Iteration 7141 / 8000) loss: 1.264629
(Time 35.02 sec; Iteration 7151 / 8000) loss: 1.170683
(Time 35.07 sec; Iteration 7161 / 8000) loss: 1.124498
(Time 35.11 sec; Iteration 7171 / 8000) loss: 1.281254
(Time 35.15 sec; Iteration 7181 / 8000) loss: 1.230934
(Time 35.19 sec; Iteration 7191 / 8000) loss: 1.052068
(Epoch 18 / 20) train acc: 0.627000; val_acc: 0.523500
(Time 35.34 sec; Iteration 7201 / 8000) loss: 0.950425
(Time 35.39 sec; Iteration 7211 / 8000) loss: 1.110284
(Time 35.43 sec; Iteration 7221 / 8000) loss: 1.154366
(Time 35.47 sec; Iteration 7231 / 8000) loss: 1.121182
(Time 35.51 sec; Iteration 7241 / 8000) loss: 1.099090
(Time 35.56 sec; Iteration 7251 / 8000) loss: 1.172849
(Time 35.60 sec; Iteration 7261 / 8000) loss: 1.050562
(Time 35.65 sec; Iteration 7271 / 8000) loss: 1.127165
(Time 35.69 sec; Iteration 7281 / 8000) loss: 0.959455
(Time 35.73 sec; Iteration 7291 / 8000) loss: 1.214903
(Time 35.77 sec; Iteration 7301 / 8000) loss: 1.192749
(Time 35.82 sec; Iteration 7311 / 8000) loss: 1.235171
(Time 35.86 sec; Iteration 7321 / 8000) loss: 1.360503
(Time 35.90 sec; Iteration 7331 / 8000) loss: 1.082100
(Time 35.94 sec; Iteration 7341 / 8000) loss: 1.220841
(Time 35.99 sec; Iteration 7351 / 8000) loss: 1.245445
(Time 36.03 sec; Iteration 7361 / 8000) loss: 1.016539
(Time 36.07 sec; Iteration 7371 / 8000) loss: 1.219668
(Time 36.12 sec; Iteration 7381 / 8000) loss: 1.150572
(Time 36.16 sec; Iteration 7391 / 8000) loss: 1.241113
(Time 36.20 sec; Iteration 7401 / 8000) loss: 1.085364
(Time 36.25 sec; Iteration 7411 / 8000) loss: 1.237737
(Time 36.29 sec; Iteration 7421 / 8000) loss: 1.072891
(Time 36.33 sec; Iteration 7431 / 8000) loss: 1.280416
(Time 36.38 sec; Iteration 7441 / 8000) loss: 1.139609
(Time 36.42 sec; Iteration 7451 / 8000) loss: 1.109688
(Time 36.46 sec; Iteration 7461 / 8000) loss: 1.341762
(Time 36.50 sec; Iteration 7471 / 8000) loss: 0.983673
(Time 36.55 sec; Iteration 7481 / 8000) loss: 1.162268
(Time 36.59 sec; Iteration 7491 / 8000) loss: 1.264371
(Time 36.64 sec; Iteration 7501 / 8000) loss: 1.001572
(Time 36.68 sec; Iteration 7511 / 8000) loss: 0.968682
(Time 36.72 sec; Iteration 7521 / 8000) loss: 1.314246
(Time 36.77 sec; Iteration 7531 / 8000) loss: 1.001908
(Time 36.81 sec; Iteration 7541 / 8000) loss: 1.206905
```

```
(Time 36.85 sec; Iteration 7551 / 8000) loss: 1.093572
(Time 36.89 sec; Iteration 7561 / 8000) loss: 1.140373
(Time 36.94 sec; Iteration 7571 / 8000) loss: 1.018711
(Time 36.98 sec; Iteration 7581 / 8000) loss: 1.290428
(Time 37.02 sec; Iteration 7591 / 8000) loss: 1.057528
(Epoch 19 / 20) train acc: 0.602000; val_acc: 0.524000
(Time 37.17 sec; Iteration 7601 / 8000) loss: 1.271423
(Time 37.21 sec; Iteration 7611 / 8000) loss: 1.192346
(Time 37.25 sec; Iteration 7621 / 8000) loss: 1.133291
(Time 37.30 sec; Iteration 7631 / 8000) loss: 1.220565
(Time 37.34 sec; Iteration 7641 / 8000) loss: 1.148030
(Time 37.38 sec; Iteration 7651 / 8000) loss: 1.028671
(Time 37.43 sec; Iteration 7661 / 8000) loss: 1.191155
(Time 37.47 sec; Iteration 7671 / 8000) loss: 1.086887
(Time 37.51 sec; Iteration 7681 / 8000) loss: 1.027545
(Time 37.55 sec; Iteration 7691 / 8000) loss: 1.278987
(Time 37.60 sec; Iteration 7701 / 8000) loss: 1.083637
(Time 37.64 sec; Iteration 7711 / 8000) loss: 1.095324
(Time 37.69 sec; Iteration 7721 / 8000) loss: 1.052646
(Time 37.73 sec; Iteration 7731 / 8000) loss: 1.109235
(Time 37.77 sec; Iteration 7741 / 8000) loss: 1.181352
(Time 37.82 sec; Iteration 7751 / 8000) loss: 1.024596
(Time 37.86 sec; Iteration 7761 / 8000) loss: 1.122099
(Time 37.90 sec; Iteration 7771 / 8000) loss: 1.157296
(Time 37.95 sec; Iteration 7781 / 8000) loss: 1.123909
(Time 37.99 sec; Iteration 7791 / 8000) loss: 1.217828
(Time 38.04 sec; Iteration 7801 / 8000) loss: 1.031844
(Time 38.08 sec; Iteration 7811 / 8000) loss: 1.185840
(Time 38.12 sec; Iteration 7821 / 8000) loss: 1.149443
(Time 38.16 sec; Iteration 7831 / 8000) loss: 1.080554
(Time 38.21 sec; Iteration 7841 / 8000) loss: 1.062056
(Time 38.25 sec; Iteration 7851 / 8000) loss: 0.950382
(Time 38.29 sec; Iteration 7861 / 8000) loss: 1.127753
(Time 38.34 sec; Iteration 7871 / 8000) loss: 0.933684
(Time 38.38 sec; Iteration 7881 / 8000) loss: 1.178936
(Time 38.42 sec; Iteration 7891 / 8000) loss: 1.198274
(Time 38.46 sec; Iteration 7901 / 8000) loss: 1.156807
(Time 38.51 sec; Iteration 7911 / 8000) loss: 1.114287
(Time 38.55 sec; Iteration 7921 / 8000) loss: 1.234844
(Time 38.59 sec; Iteration 7931 / 8000) loss: 1.126765
(Time 38.63 sec; Iteration 7941 / 8000) loss: 1.230925
(Time 38.68 sec; Iteration 7951 / 8000) loss: 1.185593
(Time 38.73 sec; Iteration 7961 / 8000) loss: 1.087695
(Time 38.77 sec; Iteration 7971 / 8000) loss: 1.145586
(Time 38.81 sec; Iteration 7981 / 8000) loss: 1.047354
(Time 38.85 sec; Iteration 7991 / 8000) loss: 1.010469
(Epoch 20 / 20) train acc: 0.642000; val_acc: 0.522200
(Time 0.00 sec; Iteration 1 / 8000) loss: 1.028362
```

```
(Epoch 20 / 20) train acc: 0.614000; val_acc: 0.522200
(Time 0.15 sec; Iteration 11 / 8000) loss: 1.187893
(Time 0.20 sec; Iteration 21 / 8000) loss: 1.278114
(Time 0.24 sec; Iteration 31 / 8000) loss: 1.139571
(Time 0.28 sec; Iteration 41 / 8000) loss: 1.306447
(Time 0.32 sec; Iteration 51 / 8000) loss: 1.168958
(Time 0.36 sec; Iteration 61 / 8000) loss: 1.166926
(Time 0.41 sec; Iteration 71 / 8000) loss: 1.063175
(Time 0.45 sec; Iteration 81 / 8000) loss: 1.102184
(Time 0.49 sec; Iteration 91 / 8000) loss: 1.255938
(Time 0.54 sec; Iteration 101 / 8000) loss: 1.030476
(Time 0.58 sec; Iteration 111 / 8000) loss: 1.053000
(Time 0.62 sec; Iteration 121 / 8000) loss: 1.078168
(Time 0.66 sec; Iteration 131 / 8000) loss: 1.135615
(Time 0.71 sec; Iteration 141 / 8000) loss: 1.134458
(Time 0.75 sec; Iteration 151 / 8000) loss: 1.175578
(Time 0.79 sec; Iteration 161 / 8000) loss: 1.123267
(Time 0.84 sec; Iteration 171 / 8000) loss: 1.227523
(Time 0.88 sec; Iteration 181 / 8000) loss: 1.090846
(Time 0.92 sec; Iteration 191 / 8000) loss: 1.153865
(Time 0.96 sec; Iteration 201 / 8000) loss: 1.107868
(Time 1.01 sec; Iteration 211 / 8000) loss: 1.166443
(Time 1.05 sec; Iteration 221 / 8000) loss: 1.206588
(Time 1.09 sec; Iteration 231 / 8000) loss: 1.126025
(Time 1.14 sec; Iteration 241 / 8000) loss: 1.122553
(Time 1.18 sec; Iteration 251 / 8000) loss: 0.994468
(Time 1.22 sec; Iteration 261 / 8000) loss: 1.048227
(Time 1.26 sec; Iteration 271 / 8000) loss: 1.214374
(Time 1.31 sec; Iteration 281 / 8000) loss: 1.094730
(Time 1.35 sec; Iteration 291 / 8000) loss: 0.950022
(Time 1.39 sec; Iteration 301 / 8000) loss: 1.102285
(Time 1.43 sec; Iteration 311 / 8000) loss: 1.336508
(Time 1.48 sec; Iteration 321 / 8000) loss: 0.996322
(Time 1.52 sec; Iteration 331 / 8000) loss: 1.048735
(Time 1.56 sec; Iteration 341 / 8000) loss: 1.054050
(Time 1.61 sec; Iteration 351 / 8000) loss: 1.193908
(Time 1.65 sec; Iteration 361 / 8000) loss: 0.962726
(Time 1.69 sec; Iteration 371 / 8000) loss: 1.147947
(Time 1.74 sec; Iteration 381 / 8000) loss: 1.025810
(Time 1.78 sec; Iteration 391 / 8000) loss: 1.184689
(Epoch 21 / 20) train acc: 0.616000; val_acc: 0.524800
(Time 1.93 sec; Iteration 401 / 8000) loss: 1.057949
(Time 1.97 sec; Iteration 411 / 8000) loss: 1.267168
(Time 2.02 sec; Iteration 421 / 8000) loss: 1.027102
(Time 2.06 sec; Iteration 431 / 8000) loss: 1.159853
(Time 2.10 sec; Iteration 441 / 8000) loss: 1.181440
(Time 2.14 sec; Iteration 451 / 8000) loss: 0.980134
(Time 2.19 sec; Iteration 461 / 8000) loss: 1.112325
```

```
(Time 2.23 sec; Iteration 471 / 8000) loss: 1.232613
(Time 2.27 sec; Iteration 481 / 8000) loss: 1.168103
(Time 2.31 sec; Iteration 491 / 8000) loss: 1.067731
(Time 2.36 sec; Iteration 501 / 8000) loss: 1.095084
(Time 2.40 sec; Iteration 511 / 8000) loss: 1.067885
(Time 2.44 sec; Iteration 521 / 8000) loss: 1.332980
(Time 2.48 sec; Iteration 531 / 8000) loss: 0.951684
(Time 2.53 sec; Iteration 541 / 8000) loss: 0.999856
(Time 2.57 sec; Iteration 551 / 8000) loss: 1.112329
(Time 2.61 sec; Iteration 561 / 8000) loss: 1.111806
(Time 2.66 sec; Iteration 571 / 8000) loss: 1.292792
(Time 2.70 sec; Iteration 581 / 8000) loss: 1.295495
(Time 2.74 sec; Iteration 591 / 8000) loss: 0.892991
(Time 2.78 sec; Iteration 601 / 8000) loss: 1.065898
(Time 2.83 sec; Iteration 611 / 8000) loss: 1.013607
(Time 2.87 sec; Iteration 621 / 8000) loss: 1.231975
(Time 2.91 sec; Iteration 631 / 8000) loss: 0.989649
(Time 2.96 sec; Iteration 641 / 8000) loss: 1.153059
(Time 3.00 sec; Iteration 651 / 8000) loss: 1.172538
(Time 3.04 sec; Iteration 661 / 8000) loss: 1.090316
(Time 3.10 sec; Iteration 671 / 8000) loss: 1.092224
(Time 3.14 sec; Iteration 681 / 8000) loss: 1.239846
(Time 3.20 sec; Iteration 691 / 8000) loss: 0.868697
(Time 3.25 sec; Iteration 701 / 8000) loss: 1.045791
(Time 3.30 sec; Iteration 711 / 8000) loss: 1.072777
(Time 3.35 sec; Iteration 721 / 8000) loss: 1.086417
(Time 3.40 sec; Iteration 731 / 8000) loss: 1.244758
(Time 3.44 sec; Iteration 741 / 8000) loss: 1.207304
(Time 3.49 sec; Iteration 751 / 8000) loss: 1.278083
(Time 3.54 sec; Iteration 761 / 8000) loss: 1.195914
(Time 3.59 sec; Iteration 771 / 8000) loss: 1.062222
(Time 3.64 sec; Iteration 781 / 8000) loss: 1.153490
(Time 3.68 sec; Iteration 791 / 8000) loss: 1.248312
(Epoch 22 / 20) train acc: 0.642000; val_acc: 0.525500
(Time 3.84 sec; Iteration 801 / 8000) loss: 1.100803
(Time 3.89 sec; Iteration 811 / 8000) loss: 1.039670
(Time 3.94 sec; Iteration 821 / 8000) loss: 1.138626
(Time 3.99 sec; Iteration 831 / 8000) loss: 1.107350
(Time 4.03 sec; Iteration 841 / 8000) loss: 0.959422
(Time 4.08 sec; Iteration 851 / 8000) loss: 1.197181
(Time 4.13 sec; Iteration 861 / 8000) loss: 1.027968
(Time 4.18 sec; Iteration 871 / 8000) loss: 1.138832
(Time 4.22 sec; Iteration 881 / 8000) loss: 1.066538
(Time 4.27 sec; Iteration 891 / 8000) loss: 1.132027
(Time 4.32 sec; Iteration 901 / 8000) loss: 1.065249
(Time 4.37 sec; Iteration 911 / 8000) loss: 1.130264
(Time 4.42 sec; Iteration 921 / 8000) loss: 1.201613
(Time 4.47 sec; Iteration 931 / 8000) loss: 1.037185
```

```
(Time 4.52 sec; Iteration 941 / 8000) loss: 0.966346
(Time 4.57 sec; Iteration 951 / 8000) loss: 0.981355
(Time 4.62 sec; Iteration 961 / 8000) loss: 0.857696
(Time 4.67 sec; Iteration 971 / 8000) loss: 0.987015
(Time 4.72 sec; Iteration 981 / 8000) loss: 1.114337
(Time 4.76 sec; Iteration 991 / 8000) loss: 1.017850
(Time 4.82 sec; Iteration 1001 / 8000) loss: 1.077129
(Time 4.86 sec; Iteration 1011 / 8000) loss: 0.917080
(Time 4.91 sec; Iteration 1021 / 8000) loss: 1.090066
(Time 4.96 sec; Iteration 1031 / 8000) loss: 1.084966
(Time 5.01 sec; Iteration 1041 / 8000) loss: 1.301450
(Time 5.06 sec; Iteration 1051 / 8000) loss: 0.939395
(Time 5.11 sec; Iteration 1061 / 8000) loss: 1.151260
(Time 5.15 sec; Iteration 1071 / 8000) loss: 1.150799
(Time 5.20 sec; Iteration 1081 / 8000) loss: 1.024228
(Time 5.25 sec; Iteration 1091 / 8000) loss: 0.928995
(Time 5.30 sec; Iteration 1101 / 8000) loss: 1.000561
(Time 5.35 sec; Iteration 1111 / 8000) loss: 1.146076
(Time 5.40 sec; Iteration 1121 / 8000) loss: 1.041277
(Time 5.45 sec; Iteration 1131 / 8000) loss: 1.335805
(Time 5.50 sec; Iteration 1141 / 8000) loss: 1.160927
(Time 5.55 sec; Iteration 1151 / 8000) loss: 0.950815
(Time 5.60 sec; Iteration 1161 / 8000) loss: 1.215968
(Time 5.65 sec; Iteration 1171 / 8000) loss: 1.076951
(Time 5.70 sec; Iteration 1181 / 8000) loss: 1.020546
(Time 5.75 sec; Iteration 1191 / 8000) loss: 1.015624
(Epoch 23 / 20) train acc: 0.625000; val_acc: 0.524300
(Time 5.91 sec; Iteration 1201 / 8000) loss: 1.130816
(Time 5.96 sec; Iteration 1211 / 8000) loss: 0.948028
(Time 6.01 sec; Iteration 1221 / 8000) loss: 1.046776
(Time 6.05 sec; Iteration 1231 / 8000) loss: 1.066069
(Time 6.10 sec; Iteration 1241 / 8000) loss: 0.997373
(Time 6.16 sec; Iteration 1251 / 8000) loss: 0.991436
(Time 6.21 sec; Iteration 1261 / 8000) loss: 1.010613
(Time 6.26 sec; Iteration 1271 / 8000) loss: 1.099190
(Time 6.30 sec; Iteration 1281 / 8000) loss: 1.031733
(Time 6.35 sec; Iteration 1291 / 8000) loss: 1.186937
(Time 6.40 sec; Iteration 1301 / 8000) loss: 0.931768
(Time 6.45 sec; Iteration 1311 / 8000) loss: 1.124787
(Time 6.50 sec; Iteration 1321 / 8000) loss: 1.102185
(Time 6.55 sec; Iteration 1331 / 8000) loss: 1.100475
(Time 6.59 sec; Iteration 1341 / 8000) loss: 1.076177
(Time 6.63 sec; Iteration 1351 / 8000) loss: 1.108843
(Time 6.68 sec; Iteration 1361 / 8000) loss: 1.051470
(Time 6.72 sec; Iteration 1371 / 8000) loss: 1.199442
(Time 6.76 sec; Iteration 1381 / 8000) loss: 1.065604
(Time 6.81 sec; Iteration 1391 / 8000) loss: 1.142303
(Time 6.85 sec; Iteration 1401 / 8000) loss: 1.057239
```

```
(Time 6.89 sec; Iteration 1411 / 8000) loss: 1.177285
(Time 6.94 sec; Iteration 1421 / 8000) loss: 1.197794
(Time 6.98 sec; Iteration 1431 / 8000) loss: 0.950265
(Time 7.02 sec; Iteration 1441 / 8000) loss: 1.239771
(Time 7.07 sec; Iteration 1451 / 8000) loss: 1.245921
(Time 7.11 sec; Iteration 1461 / 8000) loss: 1.103534
(Time 7.15 sec; Iteration 1471 / 8000) loss: 1.112517
(Time 7.19 sec; Iteration 1481 / 8000) loss: 1.225334
(Time 7.24 sec; Iteration 1491 / 8000) loss: 1.180951
(Time 7.28 sec; Iteration 1501 / 8000) loss: 1.039900
(Time 7.32 sec; Iteration 1511 / 8000) loss: 1.011631
(Time 7.36 sec; Iteration 1521 / 8000) loss: 1.099230
(Time 7.41 sec; Iteration 1531 / 8000) loss: 1.178940
(Time 7.45 sec; Iteration 1541 / 8000) loss: 1.180556
(Time 7.49 sec; Iteration 1551 / 8000) loss: 1.062091
(Time 7.54 sec; Iteration 1561 / 8000) loss: 1.016145
(Time 7.58 sec; Iteration 1571 / 8000) loss: 0.976878
(Time 7.62 sec; Iteration 1581 / 8000) loss: 0.910453
(Time 7.66 sec; Iteration 1591 / 8000) loss: 1.181745
(Epoch 24 / 20) train acc: 0.632000; val_acc: 0.526700
(Time 7.81 sec; Iteration 1601 / 8000) loss: 1.014879
(Time 7.86 sec; Iteration 1611 / 8000) loss: 1.168224
(Time 7.90 sec; Iteration 1621 / 8000) loss: 1.279308
(Time 7.95 sec; Iteration 1631 / 8000) loss: 1.010268
(Time 7.99 sec; Iteration 1641 / 8000) loss: 1.018821
(Time 8.03 sec; Iteration 1651 / 8000) loss: 1.168978
(Time 8.07 sec; Iteration 1661 / 8000) loss: 1.064403
(Time 8.12 sec; Iteration 1671 / 8000) loss: 1.103519
(Time 8.16 sec; Iteration 1681 / 8000) loss: 1.283336
(Time 8.21 sec; Iteration 1691 / 8000) loss: 0.983750
(Time 8.25 sec; Iteration 1701 / 8000) loss: 0.993992
(Time 8.29 sec; Iteration 1711 / 8000) loss: 1.049092
(Time 8.33 sec; Iteration 1721 / 8000) loss: 1.071547
(Time 8.38 sec; Iteration 1731 / 8000) loss: 1.205492
(Time 8.42 sec; Iteration 1741 / 8000) loss: 1.117152
(Time 8.46 sec; Iteration 1751 / 8000) loss: 1.197402
(Time 8.50 sec; Iteration 1761 / 8000) loss: 1.067704
(Time 8.55 sec; Iteration 1771 / 8000) loss: 1.229755
(Time 8.59 sec; Iteration 1781 / 8000) loss: 1.177722
(Time 8.63 sec; Iteration 1791 / 8000) loss: 1.085931
(Time 8.67 sec; Iteration 1801 / 8000) loss: 1.110197
(Time 8.72 sec; Iteration 1811 / 8000) loss: 0.874520
(Time 8.76 sec; Iteration 1821 / 8000) loss: 1.152090
(Time 8.80 sec; Iteration 1831 / 8000) loss: 1.008754
(Time 8.84 sec; Iteration 1841 / 8000) loss: 1.102915
(Time 8.89 sec; Iteration 1851 / 8000) loss: 1.047550
(Time 8.93 sec; Iteration 1861 / 8000) loss: 1.027629
(Time 8.98 sec; Iteration 1871 / 8000) loss: 1.236608
```

```
(Time 9.02 sec; Iteration 1881 / 8000) loss: 1.068050
(Time 9.06 sec; Iteration 1891 / 8000) loss: 1.153327
(Time 9.10 sec; Iteration 1901 / 8000) loss: 1.120519
(Time 9.15 sec; Iteration 1911 / 8000) loss: 1.077809
(Time 9.19 sec; Iteration 1921 / 8000) loss: 1.116542
(Time 9.23 sec; Iteration 1931 / 8000) loss: 1.133141
(Time 9.28 sec; Iteration 1941 / 8000) loss: 1.180753
(Time 9.32 sec; Iteration 1951 / 8000) loss: 1.210942
(Time 9.36 sec; Iteration 1961 / 8000) loss: 1.068412
(Time 9.41 sec; Iteration 1971 / 8000) loss: 1.177215
(Time 9.45 sec; Iteration 1981 / 8000) loss: 1.055769
(Time 9.49 sec; Iteration 1991 / 8000) loss: 1.144179
(Epoch 25 / 20) train acc: 0.627000; val_acc: 0.524700
(Time 9.64 sec; Iteration 2001 / 8000) loss: 1.071975
(Time 9.68 sec; Iteration 2011 / 8000) loss: 1.079872
(Time 9.72 sec; Iteration 2021 / 8000) loss: 1.038814
(Time 9.76 sec; Iteration 2031 / 8000) loss: 1.047840
(Time 9.81 sec; Iteration 2041 / 8000) loss: 1.156387
(Time 9.85 sec; Iteration 2051 / 8000) loss: 0.982725
(Time 9.89 sec; Iteration 2061 / 8000) loss: 1.198508
(Time 9.93 sec; Iteration 2071 / 8000) loss: 1.020444
(Time 9.98 sec; Iteration 2081 / 8000) loss: 1.042001
(Time 10.02 sec; Iteration 2091 / 8000) loss: 1.202129
(Time 10.07 sec; Iteration 2101 / 8000) loss: 1.044592
(Time 10.11 sec; Iteration 2111 / 8000) loss: 1.317376
(Time 10.15 sec; Iteration 2121 / 8000) loss: 0.947703
(Time 10.19 sec; Iteration 2131 / 8000) loss: 1.048025
(Time 10.24 sec; Iteration 2141 / 8000) loss: 1.032137
(Time 10.28 sec; Iteration 2151 / 8000) loss: 1.091706
(Time 10.32 sec; Iteration 2161 / 8000) loss: 1.198342
(Time 10.36 sec; Iteration 2171 / 8000) loss: 0.991577
(Time 10.41 sec; Iteration 2181 / 8000) loss: 1.115848
(Time 10.46 sec; Iteration 2191 / 8000) loss: 0.980006
(Time 10.50 sec; Iteration 2201 / 8000) loss: 1.098581
(Time 10.55 sec; Iteration 2211 / 8000) loss: 1.076264
(Time 10.59 sec; Iteration 2221 / 8000) loss: 1.057769
(Time 10.63 sec; Iteration 2231 / 8000) loss: 1.021800
(Time 10.67 sec; Iteration 2241 / 8000) loss: 0.975959
(Time 10.72 sec; Iteration 2251 / 8000) loss: 0.984616
(Time 10.76 sec; Iteration 2261 / 8000) loss: 1.251548
(Time 10.80 sec; Iteration 2271 / 8000) loss: 1.175643
(Time 10.84 sec; Iteration 2281 / 8000) loss: 1.117277
(Time 10.89 sec; Iteration 2291 / 8000) loss: 0.989733
(Time 10.93 sec; Iteration 2301 / 8000) loss: 0.995454
(Time 10.97 sec; Iteration 2311 / 8000) loss: 0.997485
(Time 11.02 sec; Iteration 2321 / 8000) loss: 1.061240
(Time 11.06 sec; Iteration 2331 / 8000) loss: 0.952641
(Time 11.10 sec; Iteration 2341 / 8000) loss: 1.085648
```

```
(Time 11.15 sec; Iteration 2351 / 8000) loss: 1.072898
(Time 11.19 sec; Iteration 2361 / 8000) loss: 1.078043
(Time 11.23 sec; Iteration 2371 / 8000) loss: 1.010505
(Time 11.28 sec; Iteration 2381 / 8000) loss: 1.149206
(Time 11.32 sec; Iteration 2391 / 8000) loss: 1.020537
(Epoch 26 / 20) train acc: 0.630000; val_acc: 0.528400
(Time 11.47 sec; Iteration 2401 / 8000) loss: 0.997347
(Time 11.51 sec; Iteration 2411 / 8000) loss: 1.041996
(Time 11.55 sec; Iteration 2421 / 8000) loss: 1.243711
(Time 11.60 sec; Iteration 2431 / 8000) loss: 0.999342
(Time 11.64 sec; Iteration 2441 / 8000) loss: 1.213276
(Time 11.68 sec; Iteration 2451 / 8000) loss: 1.021043
(Time 11.72 sec; Iteration 2461 / 8000) loss: 1.140000
(Time 11.77 sec; Iteration 2471 / 8000) loss: 1.031766
(Time 11.81 sec; Iteration 2481 / 8000) loss: 1.015704
(Time 11.85 sec; Iteration 2491 / 8000) loss: 1.160219
(Time 11.90 sec; Iteration 2501 / 8000) loss: 1.227521
(Time 11.94 sec; Iteration 2511 / 8000) loss: 1.145311
(Time 11.98 sec; Iteration 2521 / 8000) loss: 0.924988
(Time 12.02 sec; Iteration 2531 / 8000) loss: 1.085425
(Time 12.07 sec; Iteration 2541 / 8000) loss: 1.168281
(Time 12.11 sec; Iteration 2551 / 8000) loss: 1.122914
(Time 12.15 sec; Iteration 2561 / 8000) loss: 1.158806
(Time 12.20 sec; Iteration 2571 / 8000) loss: 0.994782
(Time 12.24 sec; Iteration 2581 / 8000) loss: 0.965223
(Time 12.28 sec; Iteration 2591 / 8000) loss: 1.087997
(Time 12.33 sec; Iteration 2601 / 8000) loss: 0.939632
(Time 12.37 sec; Iteration 2611 / 8000) loss: 1.220972
(Time 12.41 sec; Iteration 2621 / 8000) loss: 1.047293
(Time 12.45 sec; Iteration 2631 / 8000) loss: 0.958212
(Time 12.50 sec; Iteration 2641 / 8000) loss: 1.169003
(Time 12.54 sec; Iteration 2651 / 8000) loss: 0.958067
(Time 12.58 sec; Iteration 2661 / 8000) loss: 1.259284
(Time 12.63 sec; Iteration 2671 / 8000) loss: 1.284532
(Time 12.67 sec; Iteration 2681 / 8000) loss: 1.072319
(Time 12.71 sec; Iteration 2691 / 8000) loss: 1.021466
(Time 12.75 sec; Iteration 2701 / 8000) loss: 1.042385
(Time 12.80 sec; Iteration 2711 / 8000) loss: 1.156335
(Time 12.84 sec; Iteration 2721 / 8000) loss: 1.141137
(Time 12.88 sec; Iteration 2731 / 8000) loss: 0.932327
(Time 12.92 sec; Iteration 2741 / 8000) loss: 0.916096
(Time 12.97 sec; Iteration 2751 / 8000) loss: 1.100414
(Time 13.01 sec; Iteration 2761 / 8000) loss: 1.030435
(Time 13.06 sec; Iteration 2771 / 8000) loss: 1.167812
(Time 13.10 sec; Iteration 2781 / 8000) loss: 1.095506
(Time 13.14 sec; Iteration 2791 / 8000) loss: 1.093054
(Epoch 27 / 20) train acc: 0.620000; val_acc: 0.526700
(Time 13.29 sec; Iteration 2801 / 8000) loss: 1.137723
```

```
(Time 13.34 sec; Iteration 2811 / 8000) loss: 1.030354
(Time 13.38 sec; Iteration 2821 / 8000) loss: 1.063183
(Time 13.42 sec; Iteration 2831 / 8000) loss: 1.258352
(Time 13.47 sec; Iteration 2841 / 8000) loss: 1.077956
(Time 13.51 sec; Iteration 2851 / 8000) loss: 1.196710
(Time 13.55 sec; Iteration 2861 / 8000) loss: 0.943408
(Time 13.59 sec; Iteration 2871 / 8000) loss: 1.185991
(Time 13.64 sec; Iteration 2881 / 8000) loss: 1.015652
(Time 13.68 sec; Iteration 2891 / 8000) loss: 1.073730
(Time 13.72 sec; Iteration 2901 / 8000) loss: 1.051383
(Time 13.76 sec; Iteration 2911 / 8000) loss: 1.039166
(Time 13.81 sec; Iteration 2921 / 8000) loss: 0.952294
(Time 13.85 sec; Iteration 2931 / 8000) loss: 0.986808
(Time 13.89 sec; Iteration 2941 / 8000) loss: 0.950509
(Time 13.94 sec; Iteration 2951 / 8000) loss: 1.181298
(Time 13.99 sec; Iteration 2961 / 8000) loss: 1.191920
(Time 14.03 sec; Iteration 2971 / 8000) loss: 1.083902
(Time 14.07 sec; Iteration 2981 / 8000) loss: 0.984844
(Time 14.12 sec; Iteration 2991 / 8000) loss: 0.995850
(Time 14.16 sec; Iteration 3001 / 8000) loss: 1.098860
(Time 14.21 sec; Iteration 3011 / 8000) loss: 1.130876
(Time 14.25 sec; Iteration 3021 / 8000) loss: 1.025045
(Time 14.29 sec; Iteration 3031 / 8000) loss: 0.918005
(Time 14.33 sec; Iteration 3041 / 8000) loss: 1.128730
(Time 14.38 sec; Iteration 3051 / 8000) loss: 1.159245
(Time 14.42 sec; Iteration 3061 / 8000) loss: 1.151218
(Time 14.47 sec; Iteration 3071 / 8000) loss: 1.247374
(Time 14.51 sec; Iteration 3081 / 8000) loss: 1.021361
(Time 14.55 sec; Iteration 3091 / 8000) loss: 1.129284
(Time 14.60 sec; Iteration 3101 / 8000) loss: 1.191946
(Time 14.64 sec; Iteration 3111 / 8000) loss: 0.957430
(Time 14.68 sec; Iteration 3121 / 8000) loss: 0.976640
(Time 14.73 sec; Iteration 3131 / 8000) loss: 1.043981
(Time 14.77 sec; Iteration 3141 / 8000) loss: 0.983048
(Time 14.81 sec; Iteration 3151 / 8000) loss: 1.168954
(Time 14.86 sec; Iteration 3161 / 8000) loss: 1.050195
(Time 14.90 sec; Iteration 3171 / 8000) loss: 0.952196
(Time 14.94 sec; Iteration 3181 / 8000) loss: 1.159350
(Time 14.98 sec; Iteration 3191 / 8000) loss: 1.074248
(Epoch 28 / 20) train acc: 0.657000; val_acc: 0.525000
(Time 15.13 sec; Iteration 3201 / 8000) loss: 1.038560
(Time 15.18 sec; Iteration 3211 / 8000) loss: 1.044237
(Time 15.22 sec; Iteration 3221 / 8000) loss: 0.959766
(Time 15.26 sec; Iteration 3231 / 8000) loss: 0.865994
(Time 15.30 sec; Iteration 3241 / 8000) loss: 1.036288
(Time 15.35 sec; Iteration 3251 / 8000) loss: 1.006233
(Time 15.39 sec; Iteration 3261 / 8000) loss: 0.954454
(Time 15.43 sec; Iteration 3271 / 8000) loss: 1.024488
```

```
(Time 15.47 sec; Iteration 3281 / 8000) loss: 1.008121
(Time 15.52 sec; Iteration 3291 / 8000) loss: 1.039055
(Time 15.56 sec; Iteration 3301 / 8000) loss: 1.229364
(Time 15.60 sec; Iteration 3311 / 8000) loss: 1.045968
(Time 15.65 sec; Iteration 3321 / 8000) loss: 1.162735
(Time 15.69 sec; Iteration 3331 / 8000) loss: 1.037007
(Time 15.73 sec; Iteration 3341 / 8000) loss: 1.041847
(Time 15.78 sec; Iteration 3351 / 8000) loss: 1.192997
(Time 15.82 sec; Iteration 3361 / 8000) loss: 1.035960
(Time 15.86 sec; Iteration 3371 / 8000) loss: 0.910314
(Time 15.91 sec; Iteration 3381 / 8000) loss: 1.104273
(Time 15.95 sec; Iteration 3391 / 8000) loss: 1.218400
(Time 15.99 sec; Iteration 3401 / 8000) loss: 1.026094
(Time 16.04 sec; Iteration 3411 / 8000) loss: 1.132503
(Time 16.08 sec; Iteration 3421 / 8000) loss: 1.193855
(Time 16.13 sec; Iteration 3431 / 8000) loss: 1.125261
(Time 16.17 sec; Iteration 3441 / 8000) loss: 0.970496
(Time 16.22 sec; Iteration 3451 / 8000) loss: 0.871097
(Time 16.26 sec; Iteration 3461 / 8000) loss: 0.937893
(Time 16.30 sec; Iteration 3471 / 8000) loss: 1.037972
(Time 16.35 sec; Iteration 3481 / 8000) loss: 1.016343
(Time 16.39 sec; Iteration 3491 / 8000) loss: 1.018645
(Time 16.43 sec; Iteration 3501 / 8000) loss: 1.105636
(Time 16.47 sec; Iteration 3511 / 8000) loss: 1.014435
(Time 16.52 sec; Iteration 3521 / 8000) loss: 1.145665
(Time 16.57 sec; Iteration 3531 / 8000) loss: 1.152134
(Time 16.62 sec; Iteration 3541 / 8000) loss: 1.092020
(Time 16.67 sec; Iteration 3551 / 8000) loss: 0.934259
(Time 16.72 sec; Iteration 3561 / 8000) loss: 1.075812
(Time 16.76 sec; Iteration 3571 / 8000) loss: 1.139166
(Time 16.81 sec; Iteration 3581 / 8000) loss: 1.171488
(Time 16.86 sec; Iteration 3591 / 8000) loss: 0.961555
(Epoch 29 / 20) train acc: 0.624000; val_acc: 0.528900
(Time 17.02 sec; Iteration 3601 / 8000) loss: 0.956585
(Time 17.07 sec; Iteration 3611 / 8000) loss: 1.066383
(Time 17.11 sec; Iteration 3621 / 8000) loss: 0.982709
(Time 17.16 sec; Iteration 3631 / 8000) loss: 0.923263
(Time 17.21 sec; Iteration 3641 / 8000) loss: 1.031672
(Time 17.26 sec; Iteration 3651 / 8000) loss: 1.048213
(Time 17.31 sec; Iteration 3661 / 8000) loss: 0.973047
(Time 17.36 sec; Iteration 3671 / 8000) loss: 1.011016
(Time 17.40 sec; Iteration 3681 / 8000) loss: 1.144879
(Time 17.45 sec; Iteration 3691 / 8000) loss: 1.004933
(Time 17.50 sec; Iteration 3701 / 8000) loss: 0.981206
(Time 17.55 sec; Iteration 3711 / 8000) loss: 1.073450
(Time 17.59 sec; Iteration 3721 / 8000) loss: 1.168115
(Time 17.64 sec; Iteration 3731 / 8000) loss: 1.167557
(Time 17.69 sec; Iteration 3741 / 8000) loss: 1.065656
```

```
(Time 17.74 sec; Iteration 3751 / 8000) loss: 1.176409
(Time 17.79 sec; Iteration 3761 / 8000) loss: 0.936107
(Time 17.83 sec; Iteration 3771 / 8000) loss: 0.780957
(Time 17.88 sec; Iteration 3781 / 8000) loss: 1.031116
(Time 17.93 sec; Iteration 3791 / 8000) loss: 0.978374
(Time 17.98 sec; Iteration 3801 / 8000) loss: 1.023618
(Time 18.03 sec; Iteration 3811 / 8000) loss: 0.984905
(Time 18.08 sec; Iteration 3821 / 8000) loss: 0.977693
(Time 18.13 sec; Iteration 3831 / 8000) loss: 1.391676
(Time 18.17 sec; Iteration 3841 / 8000) loss: 1.126302
(Time 18.23 sec; Iteration 3851 / 8000) loss: 0.941164
(Time 18.28 sec; Iteration 3861 / 8000) loss: 1.013502
(Time 18.33 sec; Iteration 3871 / 8000) loss: 0.990206
(Time 18.37 sec; Iteration 3881 / 8000) loss: 1.179966
(Time 18.42 sec; Iteration 3891 / 8000) loss: 1.076730
(Time 18.47 sec; Iteration 3901 / 8000) loss: 1.154004
(Time 18.52 sec; Iteration 3911 / 8000) loss: 1.083984
(Time 18.57 sec; Iteration 3921 / 8000) loss: 1.014396
(Time 18.62 sec; Iteration 3931 / 8000) loss: 0.967754
(Time 18.66 sec; Iteration 3941 / 8000) loss: 0.997635
(Time 18.71 sec; Iteration 3951 / 8000) loss: 1.021294
(Time 18.77 sec; Iteration 3961 / 8000) loss: 0.990481
(Time 18.81 sec; Iteration 3971 / 8000) loss: 0.959959
(Time 18.86 sec; Iteration 3981 / 8000) loss: 1.000484
(Time 18.91 sec; Iteration 3991 / 8000) loss: 1.173673
(Epoch 30 / 20) train acc: 0.654000; val_acc: 0.529400
(Time 19.08 sec; Iteration 4001 / 8000) loss: 1.064162
(Time 19.13 sec; Iteration 4011 / 8000) loss: 0.872521
(Time 19.18 sec; Iteration 4021 / 8000) loss: 1.108644
(Time 19.24 sec; Iteration 4031 / 8000) loss: 1.147245
(Time 19.29 sec; Iteration 4041 / 8000) loss: 1.341501
(Time 19.34 sec; Iteration 4051 / 8000) loss: 1.177173
(Time 19.40 sec; Iteration 4061 / 8000) loss: 1.214536
(Time 19.45 sec; Iteration 4071 / 8000) loss: 0.971203
(Time 19.50 sec; Iteration 4081 / 8000) loss: 1.036689
(Time 19.55 sec; Iteration 4091 / 8000) loss: 1.168032
(Time 19.60 sec; Iteration 4101 / 8000) loss: 1.047316
(Time 19.65 sec; Iteration 4111 / 8000) loss: 1.239417
(Time 19.70 sec; Iteration 4121 / 8000) loss: 1.071869
(Time 19.75 sec; Iteration 4131 / 8000) loss: 1.171399
(Time 19.80 sec; Iteration 4141 / 8000) loss: 0.992782
(Time 19.85 sec; Iteration 4151 / 8000) loss: 1.051056
(Time 19.90 sec; Iteration 4161 / 8000) loss: 1.045443
(Time 19.95 sec; Iteration 4171 / 8000) loss: 0.892307
(Time 20.00 sec; Iteration 4181 / 8000) loss: 1.006461
(Time 20.06 sec; Iteration 4191 / 8000) loss: 1.076368
(Time 20.10 sec; Iteration 4201 / 8000) loss: 1.176648
(Time 20.14 sec; Iteration 4211 / 8000) loss: 0.975911
```

```
(Time 20.18 sec; Iteration 4221 / 8000) loss: 1.065525
(Time 20.23 sec; Iteration 4231 / 8000) loss: 1.083660
(Time 20.27 sec; Iteration 4241 / 8000) loss: 1.175123
(Time 20.32 sec; Iteration 4251 / 8000) loss: 1.105095
(Time 20.36 sec; Iteration 4261 / 8000) loss: 0.995581
(Time 20.40 sec; Iteration 4271 / 8000) loss: 1.104692
(Time 20.45 sec; Iteration 4281 / 8000) loss: 1.020542
(Time 20.49 sec; Iteration 4291 / 8000) loss: 1.136594
(Time 20.53 sec; Iteration 4301 / 8000) loss: 1.117621
(Time 20.58 sec; Iteration 4311 / 8000) loss: 1.053850
(Time 20.62 sec; Iteration 4321 / 8000) loss: 1.042553
(Time 20.66 sec; Iteration 4331 / 8000) loss: 1.085623
(Time 20.70 sec; Iteration 4341 / 8000) loss: 1.020117
(Time 20.75 sec; Iteration 4351 / 8000) loss: 0.962381
(Time 20.79 sec; Iteration 4361 / 8000) loss: 0.966122
(Time 20.83 sec; Iteration 4371 / 8000) loss: 1.128483
(Time 20.87 sec; Iteration 4381 / 8000) loss: 1.056750
(Time 20.92 sec; Iteration 4391 / 8000) loss: 1.177183
(Epoch 31 / 20) train acc: 0.645000; val_acc: 0.527300
(Time 21.06 sec; Iteration 4401 / 8000) loss: 1.055684
(Time 21.11 sec; Iteration 4411 / 8000) loss: 0.980456
(Time 21.15 sec; Iteration 4421 / 8000) loss: 1.049387
(Time 21.19 sec; Iteration 4431 / 8000) loss: 0.943774
(Time 21.24 sec; Iteration 4441 / 8000) loss: 0.991115
(Time 21.28 sec; Iteration 4451 / 8000) loss: 1.014584
(Time 21.33 sec; Iteration 4461 / 8000) loss: 1.080847
(Time 21.37 sec; Iteration 4471 / 8000) loss: 1.205095
(Time 21.41 sec; Iteration 4481 / 8000) loss: 1.101418
(Time 21.46 sec; Iteration 4491 / 8000) loss: 0.999842
(Time 21.50 sec; Iteration 4501 / 8000) loss: 0.965627
(Time 21.54 sec; Iteration 4511 / 8000) loss: 1.017952
(Time 21.58 sec; Iteration 4521 / 8000) loss: 1.101534
(Time 21.63 sec; Iteration 4531 / 8000) loss: 1.072106
(Time 21.67 sec; Iteration 4541 / 8000) loss: 1.112149
(Time 21.71 sec; Iteration 4551 / 8000) loss: 1.013411
(Time 21.75 sec; Iteration 4561 / 8000) loss: 0.973235
(Time 21.80 sec; Iteration 4571 / 8000) loss: 0.899936
(Time 21.84 sec; Iteration 4581 / 8000) loss: 1.167286
(Time 21.88 sec; Iteration 4591 / 8000) loss: 1.226838
(Time 21.92 sec; Iteration 4601 / 8000) loss: 1.189907
(Time 21.97 sec; Iteration 4611 / 8000) loss: 1.212349
(Time 22.01 sec; Iteration 4621 / 8000) loss: 1.000220
(Time 22.05 sec; Iteration 4631 / 8000) loss: 1.078083
(Time 22.10 sec; Iteration 4641 / 8000) loss: 1.160439
(Time 22.14 sec; Iteration 4651 / 8000) loss: 1.153343
(Time 22.18 sec; Iteration 4661 / 8000) loss: 0.965248
(Time 22.23 sec; Iteration 4671 / 8000) loss: 1.063297
(Time 22.27 sec; Iteration 4681 / 8000) loss: 1.149498
```

```
(Time 22.31 sec; Iteration 4691 / 8000) loss: 1.089357
(Time 22.36 sec; Iteration 4701 / 8000) loss: 0.960490
(Time 22.40 sec; Iteration 4711 / 8000) loss: 1.047774
(Time 22.44 sec; Iteration 4721 / 8000) loss: 1.042991
(Time 22.48 sec; Iteration 4731 / 8000) loss: 1.053495
(Time 22.53 sec; Iteration 4741 / 8000) loss: 1.075867
(Time 22.57 sec; Iteration 4751 / 8000) loss: 1.170817
(Time 22.61 sec; Iteration 4761 / 8000) loss: 1.126836
(Time 22.66 sec; Iteration 4771 / 8000) loss: 1.136644
(Time 22.70 sec; Iteration 4781 / 8000) loss: 0.896353
(Time 22.74 sec; Iteration 4791 / 8000) loss: 0.986386
(Epoch 32 / 20) train acc: 0.666000; val_acc: 0.529500
(Time 22.89 sec; Iteration 4801 / 8000) loss: 1.229258
(Time 22.93 sec; Iteration 4811 / 8000) loss: 1.050921
(Time 22.97 sec; Iteration 4821 / 8000) loss: 0.989474
(Time 23.02 sec; Iteration 4831 / 8000) loss: 1.068210
(Time 23.06 sec; Iteration 4841 / 8000) loss: 0.988048
(Time 23.10 sec; Iteration 4851 / 8000) loss: 1.009043
(Time 23.15 sec; Iteration 4861 / 8000) loss: 1.112315
(Time 23.19 sec; Iteration 4871 / 8000) loss: 1.095177
(Time 23.23 sec; Iteration 4881 / 8000) loss: 1.257243
(Time 23.27 sec; Iteration 4891 / 8000) loss: 1.002969
(Time 23.32 sec; Iteration 4901 / 8000) loss: 0.777319
(Time 23.36 sec; Iteration 4911 / 8000) loss: 1.086184
(Time 23.41 sec; Iteration 4921 / 8000) loss: 0.991875
(Time 23.45 sec; Iteration 4931 / 8000) loss: 1.038058
(Time 23.49 sec; Iteration 4941 / 8000) loss: 0.988522
(Time 23.54 sec; Iteration 4951 / 8000) loss: 1.041319
(Time 23.58 sec; Iteration 4961 / 8000) loss: 1.016148
(Time 23.62 sec; Iteration 4971 / 8000) loss: 1.082208
(Time 23.66 sec; Iteration 4981 / 8000) loss: 1.059120
(Time 23.71 sec; Iteration 4991 / 8000) loss: 0.918127
(Time 23.75 sec; Iteration 5001 / 8000) loss: 1.052179
(Time 23.79 sec; Iteration 5011 / 8000) loss: 0.966165
(Time 23.84 sec; Iteration 5021 / 8000) loss: 1.152386
(Time 23.88 sec; Iteration 5031 / 8000) loss: 1.110682
(Time 23.92 sec; Iteration 5041 / 8000) loss: 1.084469
(Time 23.96 sec; Iteration 5051 / 8000) loss: 1.148384
(Time 24.01 sec; Iteration 5061 / 8000) loss: 0.986756
(Time 24.05 sec; Iteration 5071 / 8000) loss: 1.148029
(Time 24.09 sec; Iteration 5081 / 8000) loss: 1.124611
(Time 24.14 sec; Iteration 5091 / 8000) loss: 0.884657
(Time 24.18 sec; Iteration 5101 / 8000) loss: 1.103886
(Time 24.22 sec; Iteration 5111 / 8000) loss: 1.052354
(Time 24.27 sec; Iteration 5121 / 8000) loss: 1.017544
(Time 24.31 sec; Iteration 5131 / 8000) loss: 1.219006
(Time 24.35 sec; Iteration 5141 / 8000) loss: 1.029897
(Time 24.40 sec; Iteration 5151 / 8000) loss: 0.855786
```

```
(Time 24.44 sec; Iteration 5161 / 8000) loss: 1.098795
(Time 24.48 sec; Iteration 5171 / 8000) loss: 1.103749
(Time 24.53 sec; Iteration 5181 / 8000) loss: 0.880212
(Time 24.57 sec; Iteration 5191 / 8000) loss: 1.115499
(Epoch 33 / 20) train acc: 0.649000; val_acc: 0.530400
(Time 24.72 sec; Iteration 5201 / 8000) loss: 0.944723
(Time 24.76 sec; Iteration 5211 / 8000) loss: 1.216088
(Time 24.80 sec; Iteration 5221 / 8000) loss: 1.009959
(Time 24.84 sec; Iteration 5231 / 8000) loss: 1.078347
(Time 24.89 sec; Iteration 5241 / 8000) loss: 1.054079
(Time 24.93 sec; Iteration 5251 / 8000) loss: 0.963157
(Time 24.97 sec; Iteration 5261 / 8000) loss: 1.058692
(Time 25.02 sec; Iteration 5271 / 8000) loss: 1.247661
(Time 25.06 sec; Iteration 5281 / 8000) loss: 1.112308
(Time 25.10 sec; Iteration 5291 / 8000) loss: 1.075897
(Time 25.15 sec; Iteration 5301 / 8000) loss: 1.126631
(Time 25.19 sec; Iteration 5311 / 8000) loss: 1.136316
(Time 25.23 sec; Iteration 5321 / 8000) loss: 1.008239
(Time 25.27 sec; Iteration 5331 / 8000) loss: 1.008455
(Time 25.32 sec; Iteration 5341 / 8000) loss: 0.933461
(Time 25.36 sec; Iteration 5351 / 8000) loss: 1.045788
(Time 25.41 sec; Iteration 5361 / 8000) loss: 1.041973
(Time 25.45 sec; Iteration 5371 / 8000) loss: 1.163808
(Time 25.49 sec; Iteration 5381 / 8000) loss: 1.080235
(Time 25.53 sec; Iteration 5391 / 8000) loss: 1.067040
(Time 25.58 sec; Iteration 5401 / 8000) loss: 0.852782
(Time 25.62 sec; Iteration 5411 / 8000) loss: 1.074566
(Time 25.66 sec; Iteration 5421 / 8000) loss: 0.977141
(Time 25.71 sec; Iteration 5431 / 8000) loss: 0.924284
(Time 25.75 sec; Iteration 5441 / 8000) loss: 1.057459
(Time 25.80 sec; Iteration 5451 / 8000) loss: 1.156421
(Time 25.84 sec; Iteration 5461 / 8000) loss: 1.156808
(Time 25.88 sec; Iteration 5471 / 8000) loss: 1.058545
(Time 25.93 sec; Iteration 5481 / 8000) loss: 0.822618
(Time 25.97 sec; Iteration 5491 / 8000) loss: 0.976899
(Time 26.02 sec; Iteration 5501 / 8000) loss: 1.087587
(Time 26.06 sec; Iteration 5511 / 8000) loss: 1.129223
(Time 26.11 sec; Iteration 5521 / 8000) loss: 0.986465
(Time 26.15 sec; Iteration 5531 / 8000) loss: 1.052268
(Time 26.19 sec; Iteration 5541 / 8000) loss: 1.156521
(Time 26.24 sec; Iteration 5551 / 8000) loss: 1.158679
(Time 26.28 sec; Iteration 5561 / 8000) loss: 1.046458
(Time 26.33 sec; Iteration 5571 / 8000) loss: 1.040633
(Time 26.37 sec; Iteration 5581 / 8000) loss: 1.100154
(Time 26.41 sec; Iteration 5591 / 8000) loss: 1.189005
(Epoch 34 / 20) train acc: 0.623000; val_acc: 0.532400
(Time 26.57 sec; Iteration 5601 / 8000) loss: 0.924078
(Time 26.61 sec; Iteration 5611 / 8000) loss: 0.846360
```

```
(Time 26.65 sec; Iteration 5621 / 8000) loss: 0.874665
(Time 26.70 sec; Iteration 5631 / 8000) loss: 1.120206
(Time 26.74 sec; Iteration 5641 / 8000) loss: 1.029389
(Time 26.78 sec; Iteration 5651 / 8000) loss: 1.040612
(Time 26.83 sec; Iteration 5661 / 8000) loss: 1.089175
(Time 26.87 sec; Iteration 5671 / 8000) loss: 1.031617
(Time 26.91 sec; Iteration 5681 / 8000) loss: 1.171954
(Time 26.95 sec; Iteration 5691 / 8000) loss: 1.005054
(Time 27.00 sec; Iteration 5701 / 8000) loss: 1.195383
(Time 27.04 sec; Iteration 5711 / 8000) loss: 0.984649
(Time 27.08 sec; Iteration 5721 / 8000) loss: 1.135544
(Time 27.13 sec; Iteration 5731 / 8000) loss: 1.099221
(Time 27.17 sec; Iteration 5741 / 8000) loss: 1.105609
(Time 27.21 sec; Iteration 5751 / 8000) loss: 1.037649
(Time 27.25 sec; Iteration 5761 / 8000) loss: 1.090707
(Time 27.30 sec; Iteration 5771 / 8000) loss: 1.023942
(Time 27.34 sec; Iteration 5781 / 8000) loss: 1.265619
(Time 27.38 sec; Iteration 5791 / 8000) loss: 1.204305
(Time 27.43 sec; Iteration 5801 / 8000) loss: 0.963659
(Time 27.48 sec; Iteration 5811 / 8000) loss: 1.096783
(Time 27.52 sec; Iteration 5821 / 8000) loss: 1.112976
(Time 27.56 sec; Iteration 5831 / 8000) loss: 1.075709
(Time 27.61 sec; Iteration 5841 / 8000) loss: 0.917021
(Time 27.66 sec; Iteration 5851 / 8000) loss: 1.012103
(Time 27.70 sec; Iteration 5861 / 8000) loss: 1.148525
(Time 27.74 sec; Iteration 5871 / 8000) loss: 1.037433
(Time 27.78 sec; Iteration 5881 / 8000) loss: 0.923727
(Time 27.83 sec; Iteration 5891 / 8000) loss: 1.154101
(Time 27.87 sec; Iteration 5901 / 8000) loss: 1.013028
(Time 27.91 sec; Iteration 5911 / 8000) loss: 1.102904
(Time 27.96 sec; Iteration 5921 / 8000) loss: 0.884304
(Time 28.00 sec; Iteration 5931 / 8000) loss: 1.203562
(Time 28.05 sec; Iteration 5941 / 8000) loss: 1.263449
(Time 28.09 sec; Iteration 5951 / 8000) loss: 1.073918
(Time 28.13 sec; Iteration 5961 / 8000) loss: 1.060117
(Time 28.18 sec; Iteration 5971 / 8000) loss: 0.961577
(Time 28.22 sec; Iteration 5981 / 8000) loss: 1.023891
(Time 28.26 sec; Iteration 5991 / 8000) loss: 1.407774
(Epoch 35 / 20) train acc: 0.656000; val_acc: 0.532200
(Time 28.41 sec; Iteration 6001 / 8000) loss: 0.855723
(Time 28.45 sec; Iteration 6011 / 8000) loss: 1.196178
(Time 28.50 sec; Iteration 6021 / 8000) loss: 1.141516
(Time 28.54 sec; Iteration 6031 / 8000) loss: 1.125299
(Time 28.59 sec; Iteration 6041 / 8000) loss: 0.984326
(Time 28.63 sec; Iteration 6051 / 8000) loss: 0.955638
(Time 28.67 sec; Iteration 6061 / 8000) loss: 1.170235
(Time 28.71 sec; Iteration 6071 / 8000) loss: 1.294259
(Time 28.76 sec; Iteration 6081 / 8000) loss: 0.962665
```

```
(Time 28.80 sec; Iteration 6091 / 8000) loss: 1.009659
(Time 28.84 sec; Iteration 6101 / 8000) loss: 1.175523
(Time 28.89 sec; Iteration 6111 / 8000) loss: 1.025441
(Time 28.93 sec; Iteration 6121 / 8000) loss: 1.155281
(Time 28.97 sec; Iteration 6131 / 8000) loss: 1.026645
(Time 29.01 sec; Iteration 6141 / 8000) loss: 1.055559
(Time 29.06 sec; Iteration 6151 / 8000) loss: 0.930236
(Time 29.10 sec; Iteration 6161 / 8000) loss: 1.015292
(Time 29.14 sec; Iteration 6171 / 8000) loss: 1.096319
(Time 29.19 sec; Iteration 6181 / 8000) loss: 1.252699
(Time 29.23 sec; Iteration 6191 / 8000) loss: 1.077014
(Time 29.27 sec; Iteration 6201 / 8000) loss: 1.033901
(Time 29.31 sec; Iteration 6211 / 8000) loss: 1.170039
(Time 29.36 sec; Iteration 6221 / 8000) loss: 0.952240
(Time 29.40 sec; Iteration 6231 / 8000) loss: 1.103701
(Time 29.45 sec; Iteration 6241 / 8000) loss: 1.120726
(Time 29.50 sec; Iteration 6251 / 8000) loss: 0.994653
(Time 29.54 sec; Iteration 6261 / 8000) loss: 1.037562
(Time 29.59 sec; Iteration 6271 / 8000) loss: 1.065984
(Time 29.63 sec; Iteration 6281 / 8000) loss: 1.176073
(Time 29.67 sec; Iteration 6291 / 8000) loss: 1.238879
(Time 29.72 sec; Iteration 6301 / 8000) loss: 1.086865
(Time 29.76 sec; Iteration 6311 / 8000) loss: 0.830385
(Time 29.80 sec; Iteration 6321 / 8000) loss: 1.186784
(Time 29.84 sec; Iteration 6331 / 8000) loss: 0.976543
(Time 29.89 sec; Iteration 6341 / 8000) loss: 0.939853
(Time 29.93 sec; Iteration 6351 / 8000) loss: 1.092792
(Time 29.97 sec; Iteration 6361 / 8000) loss: 1.081261
(Time 30.01 sec; Iteration 6371 / 8000) loss: 0.947093
(Time 30.06 sec; Iteration 6381 / 8000) loss: 1.036760
(Time 30.12 sec; Iteration 6391 / 8000) loss: 1.002587
(Epoch 36 / 20) train acc: 0.649000; val_acc: 0.532600
(Time 30.28 sec; Iteration 6401 / 8000) loss: 0.813167
(Time 30.33 sec; Iteration 6411 / 8000) loss: 1.109414
(Time 30.38 sec; Iteration 6421 / 8000) loss: 1.135626
(Time 30.43 sec; Iteration 6431 / 8000) loss: 0.983668
(Time 30.48 sec; Iteration 6441 / 8000) loss: 0.988550
(Time 30.53 sec; Iteration 6451 / 8000) loss: 1.171934
(Time 30.58 sec; Iteration 6461 / 8000) loss: 1.162661
(Time 30.63 sec; Iteration 6471 / 8000) loss: 1.015024
(Time 30.68 sec; Iteration 6481 / 8000) loss: 1.048648
(Time 30.73 sec; Iteration 6491 / 8000) loss: 0.975201
(Time 30.78 sec; Iteration 6501 / 8000) loss: 1.188773
(Time 30.83 sec; Iteration 6511 / 8000) loss: 1.035168
(Time 30.87 sec; Iteration 6521 / 8000) loss: 1.161438
(Time 30.93 sec; Iteration 6531 / 8000) loss: 1.130424
(Time 30.98 sec; Iteration 6541 / 8000) loss: 1.050438
(Time 31.02 sec; Iteration 6551 / 8000) loss: 1.040539
```

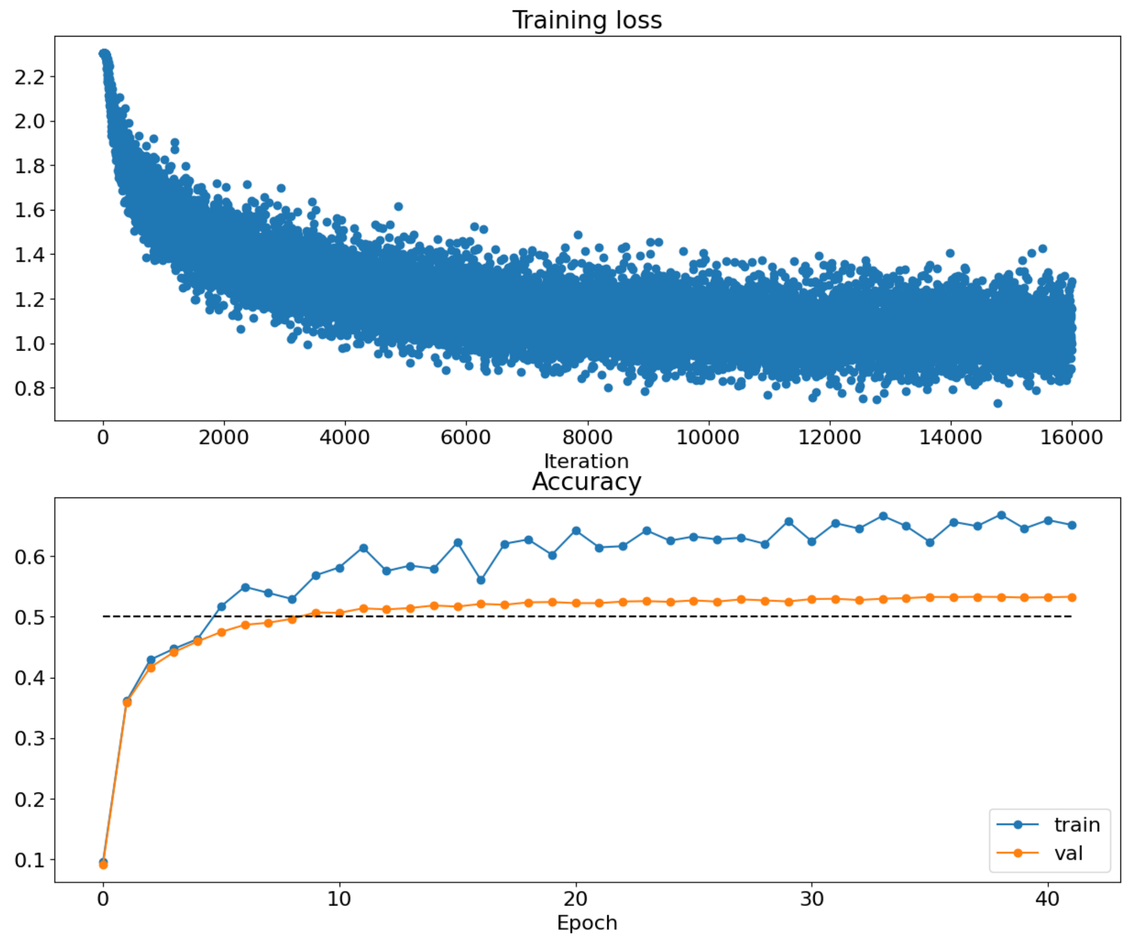
```
(Time 31.07 sec; Iteration 6561 / 8000) loss: 1.026794
(Time 31.12 sec; Iteration 6571 / 8000) loss: 0.915724
(Time 31.17 sec; Iteration 6581 / 8000) loss: 0.989606
(Time 31.22 sec; Iteration 6591 / 8000) loss: 1.074266
(Time 31.27 sec; Iteration 6601 / 8000) loss: 0.965922
(Time 31.32 sec; Iteration 6611 / 8000) loss: 0.963639
(Time 31.36 sec; Iteration 6621 / 8000) loss: 1.119051
(Time 31.42 sec; Iteration 6631 / 8000) loss: 1.016311
(Time 31.47 sec; Iteration 6641 / 8000) loss: 1.128092
(Time 31.52 sec; Iteration 6651 / 8000) loss: 1.069940
(Time 31.57 sec; Iteration 6661 / 8000) loss: 0.945783
(Time 31.62 sec; Iteration 6671 / 8000) loss: 1.083932
(Time 31.68 sec; Iteration 6681 / 8000) loss: 1.173197
(Time 31.72 sec; Iteration 6691 / 8000) loss: 1.114366
(Time 31.77 sec; Iteration 6701 / 8000) loss: 0.982822
(Time 31.82 sec; Iteration 6711 / 8000) loss: 1.069582
(Time 31.87 sec; Iteration 6721 / 8000) loss: 1.023604
(Time 31.92 sec; Iteration 6731 / 8000) loss: 0.956921
(Time 31.97 sec; Iteration 6741 / 8000) loss: 0.916165
(Time 32.02 sec; Iteration 6751 / 8000) loss: 0.993387
(Time 32.07 sec; Iteration 6761 / 8000) loss: 1.056446
(Time 32.12 sec; Iteration 6771 / 8000) loss: 0.946305
(Time 32.16 sec; Iteration 6781 / 8000) loss: 1.073521
(Time 32.21 sec; Iteration 6791 / 8000) loss: 1.298666
(Epoch 37 / 20) train acc: 0.668000; val_acc: 0.532400
(Time 32.37 sec; Iteration 6801 / 8000) loss: 0.999772
(Time 32.43 sec; Iteration 6811 / 8000) loss: 1.044145
(Time 32.48 sec; Iteration 6821 / 8000) loss: 1.054684
(Time 32.53 sec; Iteration 6831 / 8000) loss: 1.145433
(Time 32.59 sec; Iteration 6841 / 8000) loss: 1.113361
(Time 32.64 sec; Iteration 6851 / 8000) loss: 1.205861
(Time 32.69 sec; Iteration 6861 / 8000) loss: 1.113137
(Time 32.74 sec; Iteration 6871 / 8000) loss: 1.098879
(Time 32.79 sec; Iteration 6881 / 8000) loss: 1.047275
(Time 32.84 sec; Iteration 6891 / 8000) loss: 1.108062
(Time 32.89 sec; Iteration 6901 / 8000) loss: 1.105686
(Time 32.94 sec; Iteration 6911 / 8000) loss: 1.042420
(Time 32.99 sec; Iteration 6921 / 8000) loss: 1.141488
(Time 33.04 sec; Iteration 6931 / 8000) loss: 1.114809
(Time 33.09 sec; Iteration 6941 / 8000) loss: 0.979551
(Time 33.14 sec; Iteration 6951 / 8000) loss: 1.018196
(Time 33.19 sec; Iteration 6961 / 8000) loss: 0.985119
(Time 33.24 sec; Iteration 6971 / 8000) loss: 0.927890
(Time 33.28 sec; Iteration 6981 / 8000) loss: 1.067166
(Time 33.34 sec; Iteration 6991 / 8000) loss: 0.953730
(Time 33.39 sec; Iteration 7001 / 8000) loss: 1.019150
(Time 33.44 sec; Iteration 7011 / 8000) loss: 1.093767
(Time 33.49 sec; Iteration 7021 / 8000) loss: 1.105599
```

```
(Time 33.54 sec; Iteration 7031 / 8000) loss: 1.220314
(Time 33.58 sec; Iteration 7041 / 8000) loss: 1.118751
(Time 33.62 sec; Iteration 7051 / 8000) loss: 0.940614
(Time 33.67 sec; Iteration 7061 / 8000) loss: 1.041537
(Time 33.71 sec; Iteration 7071 / 8000) loss: 1.135896
(Time 33.75 sec; Iteration 7081 / 8000) loss: 1.082558
(Time 33.80 sec; Iteration 7091 / 8000) loss: 1.044241
(Time 33.84 sec; Iteration 7101 / 8000) loss: 1.024726
(Time 33.88 sec; Iteration 7111 / 8000) loss: 1.040525
(Time 33.93 sec; Iteration 7121 / 8000) loss: 0.923358
(Time 33.97 sec; Iteration 7131 / 8000) loss: 1.148237
(Time 34.01 sec; Iteration 7141 / 8000) loss: 1.063934
(Time 34.06 sec; Iteration 7151 / 8000) loss: 1.238427
(Time 34.10 sec; Iteration 7161 / 8000) loss: 1.009402
(Time 34.14 sec; Iteration 7171 / 8000) loss: 0.976684
(Time 34.18 sec; Iteration 7181 / 8000) loss: 1.212456
(Time 34.23 sec; Iteration 7191 / 8000) loss: 1.052115
(Epoch 38 / 20) train acc: 0.645000; val_acc: 0.531400
(Time 34.38 sec; Iteration 7201 / 8000) loss: 0.965501
(Time 34.42 sec; Iteration 7211 / 8000) loss: 1.149631
(Time 34.46 sec; Iteration 7221 / 8000) loss: 0.985991
(Time 34.51 sec; Iteration 7231 / 8000) loss: 0.916317
(Time 34.55 sec; Iteration 7241 / 8000) loss: 0.879772
(Time 34.59 sec; Iteration 7251 / 8000) loss: 0.992021
(Time 34.64 sec; Iteration 7261 / 8000) loss: 1.044560
(Time 34.68 sec; Iteration 7271 / 8000) loss: 1.201779
(Time 34.73 sec; Iteration 7281 / 8000) loss: 1.284894
(Time 34.77 sec; Iteration 7291 / 8000) loss: 1.069165
(Time 34.81 sec; Iteration 7301 / 8000) loss: 1.087851
(Time 34.85 sec; Iteration 7311 / 8000) loss: 1.118101
(Time 34.90 sec; Iteration 7321 / 8000) loss: 0.989644
(Time 34.94 sec; Iteration 7331 / 8000) loss: 1.069722
(Time 34.98 sec; Iteration 7341 / 8000) loss: 1.046004
(Time 35.02 sec; Iteration 7351 / 8000) loss: 1.132225
(Time 35.07 sec; Iteration 7361 / 8000) loss: 1.045392
(Time 35.11 sec; Iteration 7371 / 8000) loss: 1.188865
(Time 35.15 sec; Iteration 7381 / 8000) loss: 1.134515
(Time 35.20 sec; Iteration 7391 / 8000) loss: 0.999712
(Time 35.24 sec; Iteration 7401 / 8000) loss: 0.998445
(Time 35.28 sec; Iteration 7411 / 8000) loss: 1.067126
(Time 35.32 sec; Iteration 7421 / 8000) loss: 1.011920
(Time 35.37 sec; Iteration 7431 / 8000) loss: 1.145486
(Time 35.41 sec; Iteration 7441 / 8000) loss: 1.078773
(Time 35.45 sec; Iteration 7451 / 8000) loss: 0.977841
(Time 35.49 sec; Iteration 7461 / 8000) loss: 0.967084
(Time 35.54 sec; Iteration 7471 / 8000) loss: 1.059524
(Time 35.58 sec; Iteration 7481 / 8000) loss: 1.020078
(Time 35.62 sec; Iteration 7491 / 8000) loss: 0.968994
```

```
(Time 35.67 sec; Iteration 7501 / 8000) loss: 1.021931
(Time 35.71 sec; Iteration 7511 / 8000) loss: 1.107807
(Time 35.76 sec; Iteration 7521 / 8000) loss: 0.999001
(Time 35.80 sec; Iteration 7531 / 8000) loss: 1.041633
(Time 35.84 sec; Iteration 7541 / 8000) loss: 1.118957
(Time 35.89 sec; Iteration 7551 / 8000) loss: 1.234808
(Time 35.93 sec; Iteration 7561 / 8000) loss: 1.056436
(Time 35.97 sec; Iteration 7571 / 8000) loss: 1.027807
(Time 36.01 sec; Iteration 7581 / 8000) loss: 1.005548
(Time 36.06 sec; Iteration 7591 / 8000) loss: 1.126569
(Epoch 39 / 20) train acc: 0.659000; val_acc: 0.531800
(Time 36.21 sec; Iteration 7601 / 8000) loss: 0.956846
(Time 36.25 sec; Iteration 7611 / 8000) loss: 0.983115
(Time 36.29 sec; Iteration 7621 / 8000) loss: 1.077455
(Time 36.34 sec; Iteration 7631 / 8000) loss: 1.046629
(Time 36.38 sec; Iteration 7641 / 8000) loss: 1.043867
(Time 36.42 sec; Iteration 7651 / 8000) loss: 0.982133
(Time 36.46 sec; Iteration 7661 / 8000) loss: 1.048697
(Time 36.51 sec; Iteration 7671 / 8000) loss: 0.955246
(Time 36.55 sec; Iteration 7681 / 8000) loss: 0.980321
(Time 36.59 sec; Iteration 7691 / 8000) loss: 1.125324
(Time 36.63 sec; Iteration 7701 / 8000) loss: 1.050219
(Time 36.68 sec; Iteration 7711 / 8000) loss: 1.062388
(Time 36.72 sec; Iteration 7721 / 8000) loss: 1.003930
(Time 36.76 sec; Iteration 7731 / 8000) loss: 0.965169
(Time 36.80 sec; Iteration 7741 / 8000) loss: 1.089325
(Time 36.85 sec; Iteration 7751 / 8000) loss: 1.065134
(Time 36.89 sec; Iteration 7761 / 8000) loss: 0.894919
(Time 36.93 sec; Iteration 7771 / 8000) loss: 1.023940
(Time 36.97 sec; Iteration 7781 / 8000) loss: 1.207703
(Time 37.02 sec; Iteration 7791 / 8000) loss: 1.232491
(Time 37.06 sec; Iteration 7801 / 8000) loss: 0.829647
(Time 37.10 sec; Iteration 7811 / 8000) loss: 0.882983
(Time 37.15 sec; Iteration 7821 / 8000) loss: 1.150743
(Time 37.19 sec; Iteration 7831 / 8000) loss: 1.019572
(Time 37.23 sec; Iteration 7841 / 8000) loss: 1.038453
(Time 37.28 sec; Iteration 7851 / 8000) loss: 1.147060
(Time 37.32 sec; Iteration 7861 / 8000) loss: 1.080238
(Time 37.36 sec; Iteration 7871 / 8000) loss: 0.955218
(Time 37.40 sec; Iteration 7881 / 8000) loss: 1.049466
(Time 37.45 sec; Iteration 7891 / 8000) loss: 0.996957
(Time 37.49 sec; Iteration 7901 / 8000) loss: 1.160773
(Time 37.53 sec; Iteration 7911 / 8000) loss: 1.080544
(Time 37.58 sec; Iteration 7921 / 8000) loss: 1.073205
(Time 37.62 sec; Iteration 7931 / 8000) loss: 1.127744
(Time 37.66 sec; Iteration 7941 / 8000) loss: 1.178385
(Time 37.71 sec; Iteration 7951 / 8000) loss: 0.969315
(Time 37.75 sec; Iteration 7961 / 8000) loss: 1.121697
```

```
(Time 37.80 sec; Iteration 7971 / 8000) loss: 1.105218
(Time 37.84 sec; Iteration 7981 / 8000) loss: 1.129158
(Time 37.88 sec; Iteration 7991 / 8000) loss: 0.888892
(Epoch 40 / 20) train acc: 0.651000; val_acc: 0.532700
```

[18]:
```
# Run this cell to visualize training loss and train / val accuracy
plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



If you're happy with the model's perfromance, run the following cell to save it.

We will also reload the model and run it on validation to verify it's the right weights.

[19]:
```
path = os.path.join(GOOGLE_DRIVE_PATH, 'best_two_layer_net.pth')
solver.model.save(path)

# Create a new instance
from fully_connected_networks import create_solver_instance
reset_seed(0)

solver = create_solver_instance(data_dict=data_dict, dtype=torch.float64,␣
 ↪device='cuda')

# Load model
solver.model.load(path, dtype=torch.float64, device='cuda')

# Evaluate on validation set
accuracy = solver.check_accuracy(solver.X_val, solver.y_val)
print(f"Saved model's accuracy on validation is {accuracy}")
```

```
Saved in drive/My Drive/ITE4052/A4/best_two_layer_net.pth
(Time 0.01 sec; Iteration 1 / 8000) loss: 2.302603
(Epoch 0 / 20) train acc: 0.096000; val_acc: 0.092200
(Time 0.16 sec; Iteration 11 / 8000) loss: 2.302131
(Time 0.20 sec; Iteration 21 / 8000) loss: 2.301974
(Time 0.24 sec; Iteration 31 / 8000) loss: 2.302352
(Time 0.28 sec; Iteration 41 / 8000) loss: 2.297661
(Time 0.33 sec; Iteration 51 / 8000) loss: 2.283626
(Time 0.37 sec; Iteration 61 / 8000) loss: 2.268614
(Time 0.41 sec; Iteration 71 / 8000) loss: 2.212975
(Time 0.45 sec; Iteration 81 / 8000) loss: 2.185241
(Time 0.50 sec; Iteration 91 / 8000) loss: 2.173307
(Time 0.54 sec; Iteration 101 / 8000) loss: 2.183683
(Time 0.59 sec; Iteration 111 / 8000) loss: 2.091396
(Time 0.63 sec; Iteration 121 / 8000) loss: 2.059497
(Time 0.67 sec; Iteration 131 / 8000) loss: 2.073542
(Time 0.72 sec; Iteration 141 / 8000) loss: 2.043060
(Time 0.76 sec; Iteration 151 / 8000) loss: 2.057305
(Time 0.80 sec; Iteration 161 / 8000) loss: 2.060444
(Time 0.85 sec; Iteration 171 / 8000) loss: 1.969404
(Time 0.89 sec; Iteration 181 / 8000) loss: 2.018749
(Time 0.93 sec; Iteration 191 / 8000) loss: 1.918732
(Time 0.97 sec; Iteration 201 / 8000) loss: 1.925371
(Time 1.01 sec; Iteration 211 / 8000) loss: 1.986776
(Time 1.06 sec; Iteration 221 / 8000) loss: 2.040569
```

```
(Time 1.10 sec; Iteration 231 / 8000) loss: 1.862027
(Time 1.15 sec; Iteration 241 / 8000) loss: 1.882783
(Time 1.19 sec; Iteration 251 / 8000) loss: 1.999209
(Time 1.25 sec; Iteration 261 / 8000) loss: 1.881364
(Time 1.29 sec; Iteration 271 / 8000) loss: 1.869433
(Time 1.34 sec; Iteration 281 / 8000) loss: 1.920360
(Time 1.38 sec; Iteration 291 / 8000) loss: 1.734718
(Time 1.42 sec; Iteration 301 / 8000) loss: 1.746224
(Time 1.47 sec; Iteration 311 / 8000) loss: 1.827956
(Time 1.51 sec; Iteration 321 / 8000) loss: 1.936580
(Time 1.56 sec; Iteration 331 / 8000) loss: 1.991148
(Time 1.60 sec; Iteration 341 / 8000) loss: 1.981817
(Time 1.64 sec; Iteration 351 / 8000) loss: 1.872227
(Time 1.68 sec; Iteration 361 / 8000) loss: 1.688998
(Time 1.73 sec; Iteration 371 / 8000) loss: 1.839584
(Time 1.77 sec; Iteration 381 / 8000) loss: 1.810971
(Time 1.81 sec; Iteration 391 / 8000) loss: 1.808983
(Epoch 1 / 20) train acc: 0.361000; val_acc: 0.359000
(Time 1.96 sec; Iteration 401 / 8000) loss: 1.866122
(Time 2.01 sec; Iteration 411 / 8000) loss: 1.905465
(Time 2.05 sec; Iteration 421 / 8000) loss: 1.730916
(Time 2.10 sec; Iteration 431 / 8000) loss: 1.752849
(Time 2.14 sec; Iteration 441 / 8000) loss: 1.692051
(Time 2.18 sec; Iteration 451 / 8000) loss: 1.627449
(Time 2.23 sec; Iteration 461 / 8000) loss: 1.766686
(Time 2.27 sec; Iteration 471 / 8000) loss: 1.754263
(Time 2.31 sec; Iteration 481 / 8000) loss: 1.680517
(Time 2.35 sec; Iteration 491 / 8000) loss: 1.819147
(Time 2.39 sec; Iteration 501 / 8000) loss: 1.848964
(Time 2.44 sec; Iteration 511 / 8000) loss: 1.664147
(Time 2.48 sec; Iteration 521 / 8000) loss: 1.861571
(Time 2.52 sec; Iteration 531 / 8000) loss: 1.566806
(Time 2.57 sec; Iteration 541 / 8000) loss: 1.747006
(Time 2.61 sec; Iteration 551 / 8000) loss: 1.634745
(Time 2.65 sec; Iteration 561 / 8000) loss: 1.684015
(Time 2.70 sec; Iteration 571 / 8000) loss: 1.847447
(Time 2.74 sec; Iteration 581 / 8000) loss: 1.796701
(Time 2.78 sec; Iteration 591 / 8000) loss: 1.698253
(Time 2.83 sec; Iteration 601 / 8000) loss: 1.626178
(Time 2.87 sec; Iteration 611 / 8000) loss: 1.793206
(Time 2.91 sec; Iteration 621 / 8000) loss: 1.764089
(Time 2.96 sec; Iteration 631 / 8000) loss: 1.776800
(Time 3.00 sec; Iteration 641 / 8000) loss: 1.736117
(Time 3.04 sec; Iteration 651 / 8000) loss: 1.686904
(Time 3.09 sec; Iteration 661 / 8000) loss: 1.697896
(Time 3.13 sec; Iteration 671 / 8000) loss: 1.786791
(Time 3.17 sec; Iteration 681 / 8000) loss: 1.563450
(Time 3.21 sec; Iteration 691 / 8000) loss: 1.567511
```

```
(Time 3.25 sec; Iteration 701 / 8000) loss: 1.760979
(Time 3.30 sec; Iteration 711 / 8000) loss: 1.665903
(Time 3.34 sec; Iteration 721 / 8000) loss: 1.715495
(Time 3.39 sec; Iteration 731 / 8000) loss: 1.478780
(Time 3.43 sec; Iteration 741 / 8000) loss: 1.593218
(Time 3.47 sec; Iteration 751 / 8000) loss: 1.464391
(Time 3.52 sec; Iteration 761 / 8000) loss: 1.576131
(Time 3.56 sec; Iteration 771 / 8000) loss: 1.585751
(Time 3.60 sec; Iteration 781 / 8000) loss: 1.493866
(Time 3.65 sec; Iteration 791 / 8000) loss: 1.659011
(Epoch 2 / 20) train acc: 0.429000; val_acc: 0.416400
(Time 3.80 sec; Iteration 801 / 8000) loss: 1.522012
(Time 3.84 sec; Iteration 811 / 8000) loss: 1.524754
(Time 3.88 sec; Iteration 821 / 8000) loss: 1.693718
(Time 3.93 sec; Iteration 831 / 8000) loss: 1.619641
(Time 3.97 sec; Iteration 841 / 8000) loss: 1.661180
(Time 4.01 sec; Iteration 851 / 8000) loss: 1.731999
(Time 4.06 sec; Iteration 861 / 8000) loss: 1.388202
(Time 4.10 sec; Iteration 871 / 8000) loss: 1.494590
(Time 4.14 sec; Iteration 881 / 8000) loss: 1.719062
(Time 4.18 sec; Iteration 891 / 8000) loss: 1.696230
(Time 4.23 sec; Iteration 901 / 8000) loss: 1.720399
(Time 4.28 sec; Iteration 911 / 8000) loss: 1.634952
(Time 4.33 sec; Iteration 921 / 8000) loss: 1.641377
(Time 4.38 sec; Iteration 931 / 8000) loss: 1.613847
(Time 4.42 sec; Iteration 941 / 8000) loss: 1.416605
(Time 4.48 sec; Iteration 951 / 8000) loss: 1.493810
(Time 4.53 sec; Iteration 961 / 8000) loss: 1.609787
(Time 4.58 sec; Iteration 971 / 8000) loss: 1.434485
(Time 4.63 sec; Iteration 981 / 8000) loss: 1.673444
(Time 4.68 sec; Iteration 991 / 8000) loss: 1.492794
(Time 4.73 sec; Iteration 1001 / 8000) loss: 1.417730
(Time 4.78 sec; Iteration 1011 / 8000) loss: 1.515685
(Time 4.83 sec; Iteration 1021 / 8000) loss: 1.690853
(Time 4.88 sec; Iteration 1031 / 8000) loss: 1.463434
(Time 4.93 sec; Iteration 1041 / 8000) loss: 1.534649
(Time 4.98 sec; Iteration 1051 / 8000) loss: 1.518368
(Time 5.03 sec; Iteration 1061 / 8000) loss: 1.572649
(Time 5.08 sec; Iteration 1071 / 8000) loss: 1.496677
(Time 5.13 sec; Iteration 1081 / 8000) loss: 1.467277
(Time 5.18 sec; Iteration 1091 / 8000) loss: 1.411377
(Time 5.23 sec; Iteration 1101 / 8000) loss: 1.508326
(Time 5.27 sec; Iteration 1111 / 8000) loss: 1.565717
(Time 5.32 sec; Iteration 1121 / 8000) loss: 1.502134
(Time 5.37 sec; Iteration 1131 / 8000) loss: 1.524817
(Time 5.42 sec; Iteration 1141 / 8000) loss: 1.462300
(Time 5.47 sec; Iteration 1151 / 8000) loss: 1.596607
(Time 5.52 sec; Iteration 1161 / 8000) loss: 1.412042
```

```
(Time 5.57 sec; Iteration 1171 / 8000) loss: 1.525309
(Time 5.62 sec; Iteration 1181 / 8000) loss: 1.389256
(Time 5.67 sec; Iteration 1191 / 8000) loss: 1.458935
(Epoch 3 / 20) train acc: 0.447000; val_acc: 0.441400
(Time 5.83 sec; Iteration 1201 / 8000) loss: 1.573176
(Time 5.88 sec; Iteration 1211 / 8000) loss: 1.467197
(Time 5.93 sec; Iteration 1221 / 8000) loss: 1.607835
(Time 5.98 sec; Iteration 1231 / 8000) loss: 1.537023
(Time 6.03 sec; Iteration 1241 / 8000) loss: 1.591737
(Time 6.07 sec; Iteration 1251 / 8000) loss: 1.525906
(Time 6.13 sec; Iteration 1261 / 8000) loss: 1.488748
(Time 6.17 sec; Iteration 1271 / 8000) loss: 1.342857
(Time 6.22 sec; Iteration 1281 / 8000) loss: 1.487766
(Time 6.27 sec; Iteration 1291 / 8000) loss: 1.296228
(Time 6.32 sec; Iteration 1301 / 8000) loss: 1.415928
(Time 6.37 sec; Iteration 1311 / 8000) loss: 1.487382
(Time 6.42 sec; Iteration 1321 / 8000) loss: 1.467810
(Time 6.47 sec; Iteration 1331 / 8000) loss: 1.560782
(Time 6.52 sec; Iteration 1341 / 8000) loss: 1.538561
(Time 6.57 sec; Iteration 1351 / 8000) loss: 1.659575
(Time 6.62 sec; Iteration 1361 / 8000) loss: 1.516713
(Time 6.67 sec; Iteration 1371 / 8000) loss: 1.435643
(Time 6.73 sec; Iteration 1381 / 8000) loss: 1.490255
(Time 6.77 sec; Iteration 1391 / 8000) loss: 1.615745
(Time 6.83 sec; Iteration 1401 / 8000) loss: 1.681612
(Time 6.88 sec; Iteration 1411 / 8000) loss: 1.311217
(Time 6.93 sec; Iteration 1421 / 8000) loss: 1.531865
(Time 6.98 sec; Iteration 1431 / 8000) loss: 1.448577
(Time 7.03 sec; Iteration 1441 / 8000) loss: 1.437073
(Time 7.08 sec; Iteration 1451 / 8000) loss: 1.616219
(Time 7.13 sec; Iteration 1461 / 8000) loss: 1.583338
(Time 7.18 sec; Iteration 1471 / 8000) loss: 1.532556
(Time 7.23 sec; Iteration 1481 / 8000) loss: 1.563625
(Time 7.28 sec; Iteration 1491 / 8000) loss: 1.614664
(Time 7.34 sec; Iteration 1501 / 8000) loss: 1.386953
(Time 7.38 sec; Iteration 1511 / 8000) loss: 1.421970
(Time 7.43 sec; Iteration 1521 / 8000) loss: 1.266908
(Time 7.48 sec; Iteration 1531 / 8000) loss: 1.407091
(Time 7.54 sec; Iteration 1541 / 8000) loss: 1.641110
(Time 7.58 sec; Iteration 1551 / 8000) loss: 1.633930
(Time 7.63 sec; Iteration 1561 / 8000) loss: 1.402136
(Time 7.68 sec; Iteration 1571 / 8000) loss: 1.354367
(Time 7.74 sec; Iteration 1581 / 8000) loss: 1.241307
(Time 7.78 sec; Iteration 1591 / 8000) loss: 1.481929
(Epoch 4 / 20) train acc: 0.463000; val_acc: 0.459100
(Time 7.93 sec; Iteration 1601 / 8000) loss: 1.521766
(Time 7.97 sec; Iteration 1611 / 8000) loss: 1.242913
(Time 8.02 sec; Iteration 1621 / 8000) loss: 1.367333
```

```
(Time 8.06 sec; Iteration 1631 / 8000) loss: 1.551271
(Time 8.11 sec; Iteration 1641 / 8000) loss: 1.547451
(Time 8.15 sec; Iteration 1651 / 8000) loss: 1.404027
(Time 8.19 sec; Iteration 1661 / 8000) loss: 1.451265
(Time 8.23 sec; Iteration 1671 / 8000) loss: 1.512043
(Time 8.28 sec; Iteration 1681 / 8000) loss: 1.411674
(Time 8.32 sec; Iteration 1691 / 8000) loss: 1.390853
(Time 8.36 sec; Iteration 1701 / 8000) loss: 1.335102
(Time 8.40 sec; Iteration 1711 / 8000) loss: 1.486694
(Time 8.45 sec; Iteration 1721 / 8000) loss: 1.555205
(Time 8.49 sec; Iteration 1731 / 8000) loss: 1.563746
(Time 8.53 sec; Iteration 1741 / 8000) loss: 1.477284
(Time 8.57 sec; Iteration 1751 / 8000) loss: 1.505864
(Time 8.62 sec; Iteration 1761 / 8000) loss: 1.271154
(Time 8.66 sec; Iteration 1771 / 8000) loss: 1.525193
(Time 8.70 sec; Iteration 1781 / 8000) loss: 1.389323
(Time 8.75 sec; Iteration 1791 / 8000) loss: 1.468372
(Time 8.79 sec; Iteration 1801 / 8000) loss: 1.417168
(Time 8.83 sec; Iteration 1811 / 8000) loss: 1.416766
(Time 8.88 sec; Iteration 1821 / 8000) loss: 1.508289
(Time 8.92 sec; Iteration 1831 / 8000) loss: 1.426444
(Time 8.96 sec; Iteration 1841 / 8000) loss: 1.395155
(Time 9.00 sec; Iteration 1851 / 8000) loss: 1.336724
(Time 9.05 sec; Iteration 1861 / 8000) loss: 1.288793
(Time 9.09 sec; Iteration 1871 / 8000) loss: 1.305221
(Time 9.13 sec; Iteration 1881 / 8000) loss: 1.337113
(Time 9.18 sec; Iteration 1891 / 8000) loss: 1.498912
(Time 9.22 sec; Iteration 1901 / 8000) loss: 1.364826
(Time 9.26 sec; Iteration 1911 / 8000) loss: 1.266581
(Time 9.31 sec; Iteration 1921 / 8000) loss: 1.352975
(Time 9.35 sec; Iteration 1931 / 8000) loss: 1.417407
(Time 9.39 sec; Iteration 1941 / 8000) loss: 1.237848
(Time 9.43 sec; Iteration 1951 / 8000) loss: 1.352375
(Time 9.47 sec; Iteration 1961 / 8000) loss: 1.264502
(Time 9.52 sec; Iteration 1971 / 8000) loss: 1.476294
(Time 9.56 sec; Iteration 1981 / 8000) loss: 1.172755
(Time 9.60 sec; Iteration 1991 / 8000) loss: 1.470435
(Epoch 5 / 20) train acc: 0.517000; val_acc: 0.474700
(Time 9.75 sec; Iteration 2001 / 8000) loss: 1.412173
(Time 9.80 sec; Iteration 2011 / 8000) loss: 1.365631
(Time 9.84 sec; Iteration 2021 / 8000) loss: 1.380066
(Time 9.88 sec; Iteration 2031 / 8000) loss: 1.388442
(Time 9.93 sec; Iteration 2041 / 8000) loss: 1.406061
(Time 9.97 sec; Iteration 2051 / 8000) loss: 1.527548
(Time 10.01 sec; Iteration 2061 / 8000) loss: 1.423839
(Time 10.05 sec; Iteration 2071 / 8000) loss: 1.351541
(Time 10.10 sec; Iteration 2081 / 8000) loss: 1.392506
(Time 10.14 sec; Iteration 2091 / 8000) loss: 1.415820
```

```
(Time 10.18 sec; Iteration 2101 / 8000) loss: 1.368111
(Time 10.23 sec; Iteration 2111 / 8000) loss: 1.495955
(Time 10.27 sec; Iteration 2121 / 8000) loss: 1.439826
(Time 10.31 sec; Iteration 2131 / 8000) loss: 1.424299
(Time 10.35 sec; Iteration 2141 / 8000) loss: 1.480144
(Time 10.40 sec; Iteration 2151 / 8000) loss: 1.535295
(Time 10.44 sec; Iteration 2161 / 8000) loss: 1.348136
(Time 10.48 sec; Iteration 2171 / 8000) loss: 1.192357
(Time 10.52 sec; Iteration 2181 / 8000) loss: 1.474535
(Time 10.56 sec; Iteration 2191 / 8000) loss: 1.315101
(Time 10.61 sec; Iteration 2201 / 8000) loss: 1.386365
(Time 10.65 sec; Iteration 2211 / 8000) loss: 1.292390
(Time 10.69 sec; Iteration 2221 / 8000) loss: 1.374970
(Time 10.74 sec; Iteration 2231 / 8000) loss: 1.224959
(Time 10.78 sec; Iteration 2241 / 8000) loss: 1.395312
(Time 10.83 sec; Iteration 2251 / 8000) loss: 1.419005
(Time 10.87 sec; Iteration 2261 / 8000) loss: 1.312549
(Time 10.91 sec; Iteration 2271 / 8000) loss: 1.373220
(Time 10.96 sec; Iteration 2281 / 8000) loss: 1.416466
(Time 11.00 sec; Iteration 2291 / 8000) loss: 1.237260
(Time 11.04 sec; Iteration 2301 / 8000) loss: 1.500228
(Time 11.08 sec; Iteration 2311 / 8000) loss: 1.371111
(Time 11.13 sec; Iteration 2321 / 8000) loss: 1.398013
(Time 11.17 sec; Iteration 2331 / 8000) loss: 1.454128
(Time 11.21 sec; Iteration 2341 / 8000) loss: 1.293662
(Time 11.25 sec; Iteration 2351 / 8000) loss: 1.318695
(Time 11.30 sec; Iteration 2361 / 8000) loss: 1.328741
(Time 11.34 sec; Iteration 2371 / 8000) loss: 1.452432
(Time 11.38 sec; Iteration 2381 / 8000) loss: 1.607329
(Time 11.42 sec; Iteration 2391 / 8000) loss: 1.353413
(Epoch 6 / 20) train acc: 0.549000; val_acc: 0.486500
(Time 11.57 sec; Iteration 2401 / 8000) loss: 1.341348
(Time 11.62 sec; Iteration 2411 / 8000) loss: 1.454511
(Time 11.66 sec; Iteration 2421 / 8000) loss: 1.530261
(Time 11.71 sec; Iteration 2431 / 8000) loss: 1.269506
(Time 11.75 sec; Iteration 2441 / 8000) loss: 1.215866
(Time 11.79 sec; Iteration 2451 / 8000) loss: 1.234394
(Time 11.84 sec; Iteration 2461 / 8000) loss: 1.497389
(Time 11.88 sec; Iteration 2471 / 8000) loss: 1.437315
(Time 11.92 sec; Iteration 2481 / 8000) loss: 1.112067
(Time 11.97 sec; Iteration 2491 / 8000) loss: 1.571989
(Time 12.01 sec; Iteration 2501 / 8000) loss: 1.347804
(Time 12.05 sec; Iteration 2511 / 8000) loss: 1.473773
(Time 12.10 sec; Iteration 2521 / 8000) loss: 1.311973
(Time 12.14 sec; Iteration 2531 / 8000) loss: 1.300632
(Time 12.18 sec; Iteration 2541 / 8000) loss: 1.323950
(Time 12.22 sec; Iteration 2551 / 8000) loss: 1.197390
(Time 12.27 sec; Iteration 2561 / 8000) loss: 1.297808
```

```
(Time 12.31 sec; Iteration 2571 / 8000) loss: 1.443908
(Time 12.35 sec; Iteration 2581 / 8000) loss: 1.362915
(Time 12.39 sec; Iteration 2591 / 8000) loss: 1.387469
(Time 12.44 sec; Iteration 2601 / 8000) loss: 1.197494
(Time 12.48 sec; Iteration 2611 / 8000) loss: 1.331915
(Time 12.52 sec; Iteration 2621 / 8000) loss: 1.407377
(Time 12.57 sec; Iteration 2631 / 8000) loss: 1.348649
(Time 12.61 sec; Iteration 2641 / 8000) loss: 1.522015
(Time 12.65 sec; Iteration 2651 / 8000) loss: 1.474020
(Time 12.69 sec; Iteration 2661 / 8000) loss: 1.439386
(Time 12.74 sec; Iteration 2671 / 8000) loss: 1.443060
(Time 12.78 sec; Iteration 2681 / 8000) loss: 1.181620
(Time 12.82 sec; Iteration 2691 / 8000) loss: 1.268568
(Time 12.87 sec; Iteration 2701 / 8000) loss: 1.278015
(Time 12.91 sec; Iteration 2711 / 8000) loss: 1.255225
(Time 12.96 sec; Iteration 2721 / 8000) loss: 1.162669
(Time 13.00 sec; Iteration 2731 / 8000) loss: 1.227514
(Time 13.04 sec; Iteration 2741 / 8000) loss: 1.356799
(Time 13.08 sec; Iteration 2751 / 8000) loss: 1.235003
(Time 13.12 sec; Iteration 2761 / 8000) loss: 1.399809
(Time 13.17 sec; Iteration 2771 / 8000) loss: 1.254974
(Time 13.21 sec; Iteration 2781 / 8000) loss: 1.417478
(Time 13.25 sec; Iteration 2791 / 8000) loss: 1.201602
(Epoch 7 / 20) train acc: 0.539000; val_acc: 0.489900
(Time 13.40 sec; Iteration 2801 / 8000) loss: 1.325908
(Time 13.45 sec; Iteration 2811 / 8000) loss: 1.259086
(Time 13.49 sec; Iteration 2821 / 8000) loss: 1.363871
(Time 13.53 sec; Iteration 2831 / 8000) loss: 1.341692
(Time 13.57 sec; Iteration 2841 / 8000) loss: 1.180890
(Time 13.62 sec; Iteration 2851 / 8000) loss: 1.124726
(Time 13.66 sec; Iteration 2861 / 8000) loss: 1.398927
(Time 13.70 sec; Iteration 2871 / 8000) loss: 1.309478
(Time 13.75 sec; Iteration 2881 / 8000) loss: 1.401368
(Time 13.79 sec; Iteration 2891 / 8000) loss: 1.377041
(Time 13.83 sec; Iteration 2901 / 8000) loss: 1.318557
(Time 13.88 sec; Iteration 2911 / 8000) loss: 1.292226
(Time 13.92 sec; Iteration 2921 / 8000) loss: 1.508426
(Time 13.96 sec; Iteration 2931 / 8000) loss: 1.409223
(Time 14.01 sec; Iteration 2941 / 8000) loss: 1.149919
(Time 14.05 sec; Iteration 2951 / 8000) loss: 1.287884
(Time 14.09 sec; Iteration 2961 / 8000) loss: 1.353141
(Time 14.13 sec; Iteration 2971 / 8000) loss: 1.368842
(Time 14.18 sec; Iteration 2981 / 8000) loss: 1.188245
(Time 14.22 sec; Iteration 2991 / 8000) loss: 1.343926
(Time 14.26 sec; Iteration 3001 / 8000) loss: 1.470120
(Time 14.31 sec; Iteration 3011 / 8000) loss: 1.311723
(Time 14.35 sec; Iteration 3021 / 8000) loss: 1.390933
(Time 14.39 sec; Iteration 3031 / 8000) loss: 1.181671
```

```
(Time 14.43 sec; Iteration 3041 / 8000) loss: 1.360569
(Time 14.48 sec; Iteration 3051 / 8000) loss: 1.254770
(Time 14.52 sec; Iteration 3061 / 8000) loss: 1.226239
(Time 14.56 sec; Iteration 3071 / 8000) loss: 1.171366
(Time 14.60 sec; Iteration 3081 / 8000) loss: 1.206862
(Time 14.65 sec; Iteration 3091 / 8000) loss: 1.231305
(Time 14.69 sec; Iteration 3101 / 8000) loss: 1.246286
(Time 14.73 sec; Iteration 3111 / 8000) loss: 1.442142
(Time 14.78 sec; Iteration 3121 / 8000) loss: 1.204228
(Time 14.82 sec; Iteration 3131 / 8000) loss: 1.289332
(Time 14.86 sec; Iteration 3141 / 8000) loss: 1.269922
(Time 14.91 sec; Iteration 3151 / 8000) loss: 1.245389
(Time 14.95 sec; Iteration 3161 / 8000) loss: 1.221220
(Time 15.00 sec; Iteration 3171 / 8000) loss: 1.161363
(Time 15.04 sec; Iteration 3181 / 8000) loss: 1.381036
(Time 15.08 sec; Iteration 3191 / 8000) loss: 1.286729
(Epoch 8 / 20) train acc: 0.529000; val_acc: 0.496400
(Time 15.23 sec; Iteration 3201 / 8000) loss: 1.180377
(Time 15.28 sec; Iteration 3211 / 8000) loss: 1.409708
(Time 15.32 sec; Iteration 3221 / 8000) loss: 1.242363
(Time 15.36 sec; Iteration 3231 / 8000) loss: 1.109249
(Time 15.40 sec; Iteration 3241 / 8000) loss: 1.294702
(Time 15.45 sec; Iteration 3251 / 8000) loss: 1.317802
(Time 15.49 sec; Iteration 3261 / 8000) loss: 1.347546
(Time 15.53 sec; Iteration 3271 / 8000) loss: 1.235995
(Time 15.57 sec; Iteration 3281 / 8000) loss: 1.380530
(Time 15.62 sec; Iteration 3291 / 8000) loss: 1.381805
(Time 15.66 sec; Iteration 3301 / 8000) loss: 1.360922
(Time 15.70 sec; Iteration 3311 / 8000) loss: 1.268379
(Time 15.75 sec; Iteration 3321 / 8000) loss: 1.218190
(Time 15.79 sec; Iteration 3331 / 8000) loss: 1.279020
(Time 15.83 sec; Iteration 3341 / 8000) loss: 1.219954
(Time 15.88 sec; Iteration 3351 / 8000) loss: 1.212037
(Time 15.92 sec; Iteration 3361 / 8000) loss: 1.360056
(Time 15.97 sec; Iteration 3371 / 8000) loss: 1.275914
(Time 16.01 sec; Iteration 3381 / 8000) loss: 1.101094
(Time 16.05 sec; Iteration 3391 / 8000) loss: 1.436274
(Time 16.09 sec; Iteration 3401 / 8000) loss: 1.390154
(Time 16.14 sec; Iteration 3411 / 8000) loss: 1.142683
(Time 16.18 sec; Iteration 3421 / 8000) loss: 1.312863
(Time 16.22 sec; Iteration 3431 / 8000) loss: 1.263322
(Time 16.26 sec; Iteration 3441 / 8000) loss: 1.453143
(Time 16.31 sec; Iteration 3451 / 8000) loss: 1.384671
(Time 16.35 sec; Iteration 3461 / 8000) loss: 1.509826
(Time 16.39 sec; Iteration 3471 / 8000) loss: 1.198599
(Time 16.44 sec; Iteration 3481 / 8000) loss: 1.238305
(Time 16.48 sec; Iteration 3491 / 8000) loss: 1.242721
(Time 16.52 sec; Iteration 3501 / 8000) loss: 1.126090
```

```
(Time 16.56 sec; Iteration 3511 / 8000) loss: 1.109522
(Time 16.60 sec; Iteration 3521 / 8000) loss: 1.393798
(Time 16.65 sec; Iteration 3531 / 8000) loss: 1.260682
(Time 16.69 sec; Iteration 3541 / 8000) loss: 1.296079
(Time 16.73 sec; Iteration 3551 / 8000) loss: 1.252848
(Time 16.78 sec; Iteration 3561 / 8000) loss: 1.365798
(Time 16.82 sec; Iteration 3571 / 8000) loss: 1.337792
(Time 16.86 sec; Iteration 3581 / 8000) loss: 1.396553
(Time 16.90 sec; Iteration 3591 / 8000) loss: 1.414233
(Epoch 9 / 20) train acc: 0.568000; val_acc: 0.506900
(Time 17.06 sec; Iteration 3601 / 8000) loss: 1.471752
(Time 17.11 sec; Iteration 3611 / 8000) loss: 1.306402
(Time 17.15 sec; Iteration 3621 / 8000) loss: 1.383040
(Time 17.20 sec; Iteration 3631 / 8000) loss: 1.384934
(Time 17.24 sec; Iteration 3641 / 8000) loss: 1.227346
(Time 17.29 sec; Iteration 3651 / 8000) loss: 1.154861
(Time 17.33 sec; Iteration 3661 / 8000) loss: 1.203838
(Time 17.37 sec; Iteration 3671 / 8000) loss: 1.268199
(Time 17.42 sec; Iteration 3681 / 8000) loss: 1.264610
(Time 17.46 sec; Iteration 3691 / 8000) loss: 1.090994
(Time 17.50 sec; Iteration 3701 / 8000) loss: 1.214736
(Time 17.54 sec; Iteration 3711 / 8000) loss: 1.398486
(Time 17.59 sec; Iteration 3721 / 8000) loss: 1.376327
(Time 17.63 sec; Iteration 3731 / 8000) loss: 1.272982
(Time 17.67 sec; Iteration 3741 / 8000) loss: 1.278452
(Time 17.72 sec; Iteration 3751 / 8000) loss: 1.165819
(Time 17.77 sec; Iteration 3761 / 8000) loss: 1.259966
(Time 17.82 sec; Iteration 3771 / 8000) loss: 1.371037
(Time 17.87 sec; Iteration 3781 / 8000) loss: 1.139055
(Time 17.92 sec; Iteration 3791 / 8000) loss: 1.253244
(Time 17.97 sec; Iteration 3801 / 8000) loss: 1.424030
(Time 18.02 sec; Iteration 3811 / 8000) loss: 1.281586
(Time 18.07 sec; Iteration 3821 / 8000) loss: 1.373167
(Time 18.12 sec; Iteration 3831 / 8000) loss: 1.254153
(Time 18.17 sec; Iteration 3841 / 8000) loss: 1.246466
(Time 18.22 sec; Iteration 3851 / 8000) loss: 1.205352
(Time 18.27 sec; Iteration 3861 / 8000) loss: 1.563120
(Time 18.32 sec; Iteration 3871 / 8000) loss: 1.237035
(Time 18.37 sec; Iteration 3881 / 8000) loss: 1.333726
(Time 18.42 sec; Iteration 3891 / 8000) loss: 1.190968
(Time 18.46 sec; Iteration 3901 / 8000) loss: 1.442500
(Time 18.51 sec; Iteration 3911 / 8000) loss: 1.306660
(Time 18.56 sec; Iteration 3921 / 8000) loss: 1.037206
(Time 18.61 sec; Iteration 3931 / 8000) loss: 1.385250
(Time 18.66 sec; Iteration 3941 / 8000) loss: 1.100189
(Time 18.70 sec; Iteration 3951 / 8000) loss: 1.295998
(Time 18.75 sec; Iteration 3961 / 8000) loss: 1.167010
(Time 18.80 sec; Iteration 3971 / 8000) loss: 1.510381
```

```
(Time 18.85 sec; Iteration 3981 / 8000) loss: 1.384203
(Time 18.90 sec; Iteration 3991 / 8000) loss: 1.292444
(Epoch 10 / 20) train acc: 0.581000; val_acc: 0.506100
(Time 19.06 sec; Iteration 4001 / 8000) loss: 1.260839
(Time 19.11 sec; Iteration 4011 / 8000) loss: 1.413273
(Time 19.15 sec; Iteration 4021 / 8000) loss: 1.337102
(Time 19.20 sec; Iteration 4031 / 8000) loss: 1.337654
(Time 19.25 sec; Iteration 4041 / 8000) loss: 1.264559
(Time 19.30 sec; Iteration 4051 / 8000) loss: 1.317628
(Time 19.35 sec; Iteration 4061 / 8000) loss: 1.186777
(Time 19.39 sec; Iteration 4071 / 8000) loss: 1.176233
(Time 19.44 sec; Iteration 4081 / 8000) loss: 1.219115
(Time 19.49 sec; Iteration 4091 / 8000) loss: 1.312596
(Time 19.53 sec; Iteration 4101 / 8000) loss: 1.343185
(Time 19.58 sec; Iteration 4111 / 8000) loss: 1.152542
(Time 19.63 sec; Iteration 4121 / 8000) loss: 1.299046
(Time 19.68 sec; Iteration 4131 / 8000) loss: 1.173922
(Time 19.72 sec; Iteration 4141 / 8000) loss: 1.147126
(Time 19.77 sec; Iteration 4151 / 8000) loss: 1.278966
(Time 19.82 sec; Iteration 4161 / 8000) loss: 1.149381
(Time 19.87 sec; Iteration 4171 / 8000) loss: 1.105889
(Time 19.92 sec; Iteration 4181 / 8000) loss: 1.243304
(Time 19.98 sec; Iteration 4191 / 8000) loss: 1.266655
(Time 20.03 sec; Iteration 4201 / 8000) loss: 1.227603
(Time 20.08 sec; Iteration 4211 / 8000) loss: 1.129975
(Time 20.13 sec; Iteration 4221 / 8000) loss: 1.188917
(Time 20.18 sec; Iteration 4231 / 8000) loss: 1.203324
(Time 20.23 sec; Iteration 4241 / 8000) loss: 1.205165
(Time 20.28 sec; Iteration 4251 / 8000) loss: 1.291823
(Time 20.33 sec; Iteration 4261 / 8000) loss: 1.357063
(Time 20.38 sec; Iteration 4271 / 8000) loss: 1.121283
(Time 20.43 sec; Iteration 4281 / 8000) loss: 1.176875
(Time 20.48 sec; Iteration 4291 / 8000) loss: 1.213360
(Time 20.53 sec; Iteration 4301 / 8000) loss: 1.179129
(Time 20.58 sec; Iteration 4311 / 8000) loss: 1.427982
(Time 20.63 sec; Iteration 4321 / 8000) loss: 1.092745
(Time 20.69 sec; Iteration 4331 / 8000) loss: 1.116980
(Time 20.73 sec; Iteration 4341 / 8000) loss: 1.146390
(Time 20.78 sec; Iteration 4351 / 8000) loss: 1.178112
(Time 20.83 sec; Iteration 4361 / 8000) loss: 1.170790
(Time 20.88 sec; Iteration 4371 / 8000) loss: 1.276004
(Time 20.93 sec; Iteration 4381 / 8000) loss: 1.017676
(Time 20.98 sec; Iteration 4391 / 8000) loss: 1.114798
(Epoch 11 / 20) train acc: 0.614000; val_acc: 0.513600
(Time 21.14 sec; Iteration 4401 / 8000) loss: 1.245968
(Time 21.20 sec; Iteration 4411 / 8000) loss: 1.232027
(Time 21.25 sec; Iteration 4421 / 8000) loss: 1.181578
(Time 21.29 sec; Iteration 4431 / 8000) loss: 1.030650
```

```
(Time 21.34 sec; Iteration 4441 / 8000) loss: 1.215218
(Time 21.38 sec; Iteration 4451 / 8000) loss: 1.120076
(Time 21.42 sec; Iteration 4461 / 8000) loss: 1.387417
(Time 21.46 sec; Iteration 4471 / 8000) loss: 1.309303
(Time 21.51 sec; Iteration 4481 / 8000) loss: 1.166170
(Time 21.55 sec; Iteration 4491 / 8000) loss: 1.187347
(Time 21.59 sec; Iteration 4501 / 8000) loss: 1.266367
(Time 21.64 sec; Iteration 4511 / 8000) loss: 1.303981
(Time 21.68 sec; Iteration 4521 / 8000) loss: 1.304114
(Time 21.72 sec; Iteration 4531 / 8000) loss: 1.226509
(Time 21.76 sec; Iteration 4541 / 8000) loss: 1.154562
(Time 21.81 sec; Iteration 4551 / 8000) loss: 1.269596
(Time 21.85 sec; Iteration 4561 / 8000) loss: 1.206417
(Time 21.89 sec; Iteration 4571 / 8000) loss: 1.096264
(Time 21.94 sec; Iteration 4581 / 8000) loss: 1.219308
(Time 21.98 sec; Iteration 4591 / 8000) loss: 1.353126
(Time 22.02 sec; Iteration 4601 / 8000) loss: 1.180343
(Time 22.07 sec; Iteration 4611 / 8000) loss: 1.113593
(Time 22.11 sec; Iteration 4621 / 8000) loss: 1.114961
(Time 22.16 sec; Iteration 4631 / 8000) loss: 1.201712
(Time 22.20 sec; Iteration 4641 / 8000) loss: 1.184255
(Time 22.25 sec; Iteration 4651 / 8000) loss: 1.199808
(Time 22.29 sec; Iteration 4661 / 8000) loss: 1.122299
(Time 22.33 sec; Iteration 4671 / 8000) loss: 1.129043
(Time 22.38 sec; Iteration 4681 / 8000) loss: 1.191901
(Time 22.42 sec; Iteration 4691 / 8000) loss: 1.311126
(Time 22.46 sec; Iteration 4701 / 8000) loss: 1.207893
(Time 22.51 sec; Iteration 4711 / 8000) loss: 1.302956
(Time 22.55 sec; Iteration 4721 / 8000) loss: 1.176415
(Time 22.59 sec; Iteration 4731 / 8000) loss: 1.245183
(Time 22.63 sec; Iteration 4741 / 8000) loss: 1.064497
(Time 22.68 sec; Iteration 4751 / 8000) loss: 1.232743
(Time 22.72 sec; Iteration 4761 / 8000) loss: 1.260703
(Time 22.76 sec; Iteration 4771 / 8000) loss: 1.179166
(Time 22.80 sec; Iteration 4781 / 8000) loss: 1.340145
(Time 22.85 sec; Iteration 4791 / 8000) loss: 1.148120
(Epoch 12 / 20) train acc: 0.575000; val_acc: 0.511900
(Time 23.00 sec; Iteration 4801 / 8000) loss: 1.184140
(Time 23.04 sec; Iteration 4811 / 8000) loss: 1.297443
(Time 23.08 sec; Iteration 4821 / 8000) loss: 1.097664
(Time 23.12 sec; Iteration 4831 / 8000) loss: 1.085454
(Time 23.17 sec; Iteration 4841 / 8000) loss: 1.179471
(Time 23.21 sec; Iteration 4851 / 8000) loss: 1.121438
(Time 23.26 sec; Iteration 4861 / 8000) loss: 1.202191
(Time 23.30 sec; Iteration 4871 / 8000) loss: 1.096466
(Time 23.34 sec; Iteration 4881 / 8000) loss: 1.262228
(Time 23.38 sec; Iteration 4891 / 8000) loss: 1.127605
(Time 23.43 sec; Iteration 4901 / 8000) loss: 1.172423
```

```
(Time 23.47 sec; Iteration 4911 / 8000) loss: 1.245361
(Time 23.51 sec; Iteration 4921 / 8000) loss: 1.069427
(Time 23.55 sec; Iteration 4931 / 8000) loss: 1.079448
(Time 23.59 sec; Iteration 4941 / 8000) loss: 1.199983
(Time 23.64 sec; Iteration 4951 / 8000) loss: 1.211033
(Time 23.68 sec; Iteration 4961 / 8000) loss: 1.218202
(Time 23.72 sec; Iteration 4971 / 8000) loss: 1.335323
(Time 23.77 sec; Iteration 4981 / 8000) loss: 1.234009
(Time 23.81 sec; Iteration 4991 / 8000) loss: 1.166646
(Time 23.85 sec; Iteration 5001 / 8000) loss: 1.115607
(Time 23.89 sec; Iteration 5011 / 8000) loss: 1.171114
(Time 23.94 sec; Iteration 5021 / 8000) loss: 1.153437
(Time 23.98 sec; Iteration 5031 / 8000) loss: 1.262027
(Time 24.02 sec; Iteration 5041 / 8000) loss: 1.235849
(Time 24.07 sec; Iteration 5051 / 8000) loss: 0.979780
(Time 24.11 sec; Iteration 5061 / 8000) loss: 1.318055
(Time 24.15 sec; Iteration 5071 / 8000) loss: 1.157569
(Time 24.20 sec; Iteration 5081 / 8000) loss: 1.152742
(Time 24.24 sec; Iteration 5091 / 8000) loss: 1.190425
(Time 24.28 sec; Iteration 5101 / 8000) loss: 1.148353
(Time 24.33 sec; Iteration 5111 / 8000) loss: 1.342662
(Time 24.37 sec; Iteration 5121 / 8000) loss: 1.199204
(Time 24.41 sec; Iteration 5131 / 8000) loss: 1.097798
(Time 24.45 sec; Iteration 5141 / 8000) loss: 1.185935
(Time 24.50 sec; Iteration 5151 / 8000) loss: 1.045423
(Time 24.54 sec; Iteration 5161 / 8000) loss: 1.139327
(Time 24.58 sec; Iteration 5171 / 8000) loss: 1.237807
(Time 24.62 sec; Iteration 5181 / 8000) loss: 1.261219
(Time 24.66 sec; Iteration 5191 / 8000) loss: 1.176166
(Epoch 13 / 20) train acc: 0.584000; val_acc: 0.514100
(Time 24.81 sec; Iteration 5201 / 8000) loss: 1.218754
(Time 24.86 sec; Iteration 5211 / 8000) loss: 0.949391
(Time 24.90 sec; Iteration 5221 / 8000) loss: 1.120123
(Time 24.94 sec; Iteration 5231 / 8000) loss: 1.209788
(Time 24.98 sec; Iteration 5241 / 8000) loss: 1.204999
(Time 25.03 sec; Iteration 5251 / 8000) loss: 1.205357
(Time 25.07 sec; Iteration 5261 / 8000) loss: 1.135834
(Time 25.12 sec; Iteration 5271 / 8000) loss: 1.198687
(Time 25.16 sec; Iteration 5281 / 8000) loss: 1.045356
(Time 25.20 sec; Iteration 5291 / 8000) loss: 1.156109
(Time 25.25 sec; Iteration 5301 / 8000) loss: 1.238660
(Time 25.29 sec; Iteration 5311 / 8000) loss: 1.066728
(Time 25.33 sec; Iteration 5321 / 8000) loss: 1.240030
(Time 25.38 sec; Iteration 5331 / 8000) loss: 1.103401
(Time 25.42 sec; Iteration 5341 / 8000) loss: 1.219692
(Time 25.46 sec; Iteration 5351 / 8000) loss: 1.090398
(Time 25.50 sec; Iteration 5361 / 8000) loss: 1.213761
(Time 25.55 sec; Iteration 5371 / 8000) loss: 1.000552
```

```
(Time 25.59 sec; Iteration 5381 / 8000) loss: 1.311897
(Time 25.63 sec; Iteration 5391 / 8000) loss: 1.118381
(Time 25.67 sec; Iteration 5401 / 8000) loss: 1.312198
(Time 25.72 sec; Iteration 5411 / 8000) loss: 1.121567
(Time 25.76 sec; Iteration 5421 / 8000) loss: 1.195857
(Time 25.80 sec; Iteration 5431 / 8000) loss: 1.206903
(Time 25.85 sec; Iteration 5441 / 8000) loss: 1.150481
(Time 25.89 sec; Iteration 5451 / 8000) loss: 1.108264
(Time 25.93 sec; Iteration 5461 / 8000) loss: 1.270943
(Time 25.98 sec; Iteration 5471 / 8000) loss: 1.103286
(Time 26.02 sec; Iteration 5481 / 8000) loss: 1.225827
(Time 26.06 sec; Iteration 5491 / 8000) loss: 1.031693
(Time 26.10 sec; Iteration 5501 / 8000) loss: 1.019815
(Time 26.15 sec; Iteration 5511 / 8000) loss: 1.187512
(Time 26.19 sec; Iteration 5521 / 8000) loss: 1.129688
(Time 26.23 sec; Iteration 5531 / 8000) loss: 1.122110
(Time 26.28 sec; Iteration 5541 / 8000) loss: 1.027762
(Time 26.32 sec; Iteration 5551 / 8000) loss: 1.179056
(Time 26.37 sec; Iteration 5561 / 8000) loss: 1.176161
(Time 26.41 sec; Iteration 5571 / 8000) loss: 1.226215
(Time 26.45 sec; Iteration 5581 / 8000) loss: 1.207482
(Time 26.49 sec; Iteration 5591 / 8000) loss: 1.186410
(Epoch 14 / 20) train acc: 0.579000; val_acc: 0.518300
(Time 26.64 sec; Iteration 5601 / 8000) loss: 1.176066
(Time 26.69 sec; Iteration 5611 / 8000) loss: 1.075066
(Time 26.73 sec; Iteration 5621 / 8000) loss: 1.092907
(Time 26.77 sec; Iteration 5631 / 8000) loss: 1.117329
(Time 26.81 sec; Iteration 5641 / 8000) loss: 1.146648
(Time 26.86 sec; Iteration 5651 / 8000) loss: 1.152727
(Time 26.90 sec; Iteration 5661 / 8000) loss: 0.879547
(Time 26.94 sec; Iteration 5671 / 8000) loss: 0.989005
(Time 26.98 sec; Iteration 5681 / 8000) loss: 1.243404
(Time 27.03 sec; Iteration 5691 / 8000) loss: 1.000434
(Time 27.07 sec; Iteration 5701 / 8000) loss: 1.033780
(Time 27.11 sec; Iteration 5711 / 8000) loss: 1.134701
(Time 27.15 sec; Iteration 5721 / 8000) loss: 1.095631
(Time 27.20 sec; Iteration 5731 / 8000) loss: 1.156347
(Time 27.24 sec; Iteration 5741 / 8000) loss: 1.157606
(Time 27.29 sec; Iteration 5751 / 8000) loss: 1.167874
(Time 27.33 sec; Iteration 5761 / 8000) loss: 1.162497
(Time 27.37 sec; Iteration 5771 / 8000) loss: 1.187374
(Time 27.41 sec; Iteration 5781 / 8000) loss: 1.033929
(Time 27.46 sec; Iteration 5791 / 8000) loss: 1.016022
(Time 27.50 sec; Iteration 5801 / 8000) loss: 1.193802
(Time 27.54 sec; Iteration 5811 / 8000) loss: 1.177333
(Time 27.58 sec; Iteration 5821 / 8000) loss: 0.965609
(Time 27.63 sec; Iteration 5831 / 8000) loss: 1.370536
(Time 27.67 sec; Iteration 5841 / 8000) loss: 1.257248
```

```
(Time 27.71 sec; Iteration 5851 / 8000) loss: 1.160465
(Time 27.76 sec; Iteration 5861 / 8000) loss: 1.045980
(Time 27.80 sec; Iteration 5871 / 8000) loss: 1.004162
(Time 27.84 sec; Iteration 5881 / 8000) loss: 1.171955
(Time 27.88 sec; Iteration 5891 / 8000) loss: 1.252793
(Time 27.93 sec; Iteration 5901 / 8000) loss: 1.311589
(Time 27.97 sec; Iteration 5911 / 8000) loss: 1.041840
(Time 28.01 sec; Iteration 5921 / 8000) loss: 1.184844
(Time 28.06 sec; Iteration 5931 / 8000) loss: 1.256646
(Time 28.10 sec; Iteration 5941 / 8000) loss: 1.181448
(Time 28.15 sec; Iteration 5951 / 8000) loss: 1.182284
(Time 28.19 sec; Iteration 5961 / 8000) loss: 1.255372
(Time 28.23 sec; Iteration 5971 / 8000) loss: 1.037218
(Time 28.28 sec; Iteration 5981 / 8000) loss: 1.161977
(Time 28.32 sec; Iteration 5991 / 8000) loss: 1.098796
(Epoch 15 / 20) train acc: 0.622000; val_acc: 0.516500
(Time 28.47 sec; Iteration 6001 / 8000) loss: 1.078514
(Time 28.52 sec; Iteration 6011 / 8000) loss: 1.158509
(Time 28.56 sec; Iteration 6021 / 8000) loss: 1.306555
(Time 28.60 sec; Iteration 6031 / 8000) loss: 1.094071
(Time 28.64 sec; Iteration 6041 / 8000) loss: 1.136284
(Time 28.69 sec; Iteration 6051 / 8000) loss: 1.226913
(Time 28.73 sec; Iteration 6061 / 8000) loss: 1.312909
(Time 28.78 sec; Iteration 6071 / 8000) loss: 1.004155
(Time 28.82 sec; Iteration 6081 / 8000) loss: 1.108671
(Time 28.86 sec; Iteration 6091 / 8000) loss: 1.120606
(Time 28.91 sec; Iteration 6101 / 8000) loss: 1.261905
(Time 28.95 sec; Iteration 6111 / 8000) loss: 1.190173
(Time 28.99 sec; Iteration 6121 / 8000) loss: 1.155047
(Time 29.04 sec; Iteration 6131 / 8000) loss: 1.269598
(Time 29.08 sec; Iteration 6141 / 8000) loss: 1.177689
(Time 29.12 sec; Iteration 6151 / 8000) loss: 1.035619
(Time 29.16 sec; Iteration 6161 / 8000) loss: 1.158220
(Time 29.21 sec; Iteration 6171 / 8000) loss: 1.211366
(Time 29.25 sec; Iteration 6181 / 8000) loss: 1.232188
(Time 29.29 sec; Iteration 6191 / 8000) loss: 1.178029
(Time 29.34 sec; Iteration 6201 / 8000) loss: 1.079571
(Time 29.38 sec; Iteration 6211 / 8000) loss: 1.223659
(Time 29.42 sec; Iteration 6221 / 8000) loss: 0.960388
(Time 29.47 sec; Iteration 6231 / 8000) loss: 1.264415
(Time 29.51 sec; Iteration 6241 / 8000) loss: 1.199141
(Time 29.55 sec; Iteration 6251 / 8000) loss: 1.177021
(Time 29.60 sec; Iteration 6261 / 8000) loss: 1.153305
(Time 29.64 sec; Iteration 6271 / 8000) loss: 0.996014
(Time 29.68 sec; Iteration 6281 / 8000) loss: 1.104129
(Time 29.73 sec; Iteration 6291 / 8000) loss: 1.165566
(Time 29.77 sec; Iteration 6301 / 8000) loss: 1.158667
(Time 29.81 sec; Iteration 6311 / 8000) loss: 1.121371
```

```
(Time 29.86 sec; Iteration 6321 / 8000) loss: 1.083281
(Time 29.90 sec; Iteration 6331 / 8000) loss: 1.046354
(Time 29.94 sec; Iteration 6341 / 8000) loss: 1.230958
(Time 29.99 sec; Iteration 6351 / 8000) loss: 1.088088
(Time 30.03 sec; Iteration 6361 / 8000) loss: 1.097740
(Time 30.07 sec; Iteration 6371 / 8000) loss: 1.148480
(Time 30.12 sec; Iteration 6381 / 8000) loss: 1.073526
(Time 30.16 sec; Iteration 6391 / 8000) loss: 1.216074
(Epoch 16 / 20) train acc: 0.560000; val_acc: 0.520900
(Time 30.31 sec; Iteration 6401 / 8000) loss: 1.252698
(Time 30.35 sec; Iteration 6411 / 8000) loss: 1.198587
(Time 30.40 sec; Iteration 6421 / 8000) loss: 1.100727
(Time 30.44 sec; Iteration 6431 / 8000) loss: 1.133145
(Time 30.48 sec; Iteration 6441 / 8000) loss: 1.255859
(Time 30.52 sec; Iteration 6451 / 8000) loss: 1.012849
(Time 30.57 sec; Iteration 6461 / 8000) loss: 1.216098
(Time 30.61 sec; Iteration 6471 / 8000) loss: 1.122756
(Time 30.65 sec; Iteration 6481 / 8000) loss: 0.926522
(Time 30.70 sec; Iteration 6491 / 8000) loss: 1.102520
(Time 30.74 sec; Iteration 6501 / 8000) loss: 0.913332
(Time 30.78 sec; Iteration 6511 / 8000) loss: 1.332486
(Time 30.82 sec; Iteration 6521 / 8000) loss: 1.121745
(Time 30.87 sec; Iteration 6531 / 8000) loss: 1.013009
(Time 30.91 sec; Iteration 6541 / 8000) loss: 1.141801
(Time 30.96 sec; Iteration 6551 / 8000) loss: 1.153953
(Time 31.00 sec; Iteration 6561 / 8000) loss: 1.113425
(Time 31.04 sec; Iteration 6571 / 8000) loss: 1.157148
(Time 31.09 sec; Iteration 6581 / 8000) loss: 1.141855
(Time 31.13 sec; Iteration 6591 / 8000) loss: 1.156055
(Time 31.17 sec; Iteration 6601 / 8000) loss: 1.018054
(Time 31.21 sec; Iteration 6611 / 8000) loss: 1.216052
(Time 31.26 sec; Iteration 6621 / 8000) loss: 1.280552
(Time 31.31 sec; Iteration 6631 / 8000) loss: 1.318749
(Time 31.37 sec; Iteration 6641 / 8000) loss: 1.130039
(Time 31.43 sec; Iteration 6651 / 8000) loss: 1.068184
(Time 31.48 sec; Iteration 6661 / 8000) loss: 1.002914
(Time 31.53 sec; Iteration 6671 / 8000) loss: 1.145888
(Time 31.58 sec; Iteration 6681 / 8000) loss: 1.240326
(Time 31.63 sec; Iteration 6691 / 8000) loss: 1.094868
(Time 31.67 sec; Iteration 6701 / 8000) loss: 1.107030
(Time 31.72 sec; Iteration 6711 / 8000) loss: 1.103841
(Time 31.77 sec; Iteration 6721 / 8000) loss: 1.276179
(Time 31.82 sec; Iteration 6731 / 8000) loss: 1.266849
(Time 31.87 sec; Iteration 6741 / 8000) loss: 1.220176
(Time 31.92 sec; Iteration 6751 / 8000) loss: 1.282358
(Time 31.97 sec; Iteration 6761 / 8000) loss: 1.009467
(Time 32.02 sec; Iteration 6771 / 8000) loss: 1.154689
(Time 32.06 sec; Iteration 6781 / 8000) loss: 1.241300
```

```
(Time 32.11 sec; Iteration 6791 / 8000) loss: 1.256572
(Epoch 17 / 20) train acc: 0.620000; val_acc: 0.519500
(Time 32.27 sec; Iteration 6801 / 8000) loss: 1.151431
(Time 32.32 sec; Iteration 6811 / 8000) loss: 1.218204
(Time 32.36 sec; Iteration 6821 / 8000) loss: 1.076329
(Time 32.41 sec; Iteration 6831 / 8000) loss: 1.044177
(Time 32.46 sec; Iteration 6841 / 8000) loss: 1.261696
(Time 32.52 sec; Iteration 6851 / 8000) loss: 1.020929
(Time 32.56 sec; Iteration 6861 / 8000) loss: 1.030695
(Time 32.61 sec; Iteration 6871 / 8000) loss: 1.088330
(Time 32.66 sec; Iteration 6881 / 8000) loss: 1.261959
(Time 32.71 sec; Iteration 6891 / 8000) loss: 1.138552
(Time 32.76 sec; Iteration 6901 / 8000) loss: 1.130234
(Time 32.81 sec; Iteration 6911 / 8000) loss: 1.170873
(Time 32.86 sec; Iteration 6921 / 8000) loss: 1.031831
(Time 32.91 sec; Iteration 6931 / 8000) loss: 1.111410
(Time 32.96 sec; Iteration 6941 / 8000) loss: 1.019988
(Time 33.01 sec; Iteration 6951 / 8000) loss: 1.175702
(Time 33.06 sec; Iteration 6961 / 8000) loss: 1.158080
(Time 33.10 sec; Iteration 6971 / 8000) loss: 1.061829
(Time 33.15 sec; Iteration 6981 / 8000) loss: 1.149335
(Time 33.20 sec; Iteration 6991 / 8000) loss: 1.104751
(Time 33.25 sec; Iteration 7001 / 8000) loss: 1.050101
(Time 33.30 sec; Iteration 7011 / 8000) loss: 1.158775
(Time 33.34 sec; Iteration 7021 / 8000) loss: 1.100383
(Time 33.39 sec; Iteration 7031 / 8000) loss: 1.041710
(Time 33.44 sec; Iteration 7041 / 8000) loss: 1.111438
(Time 33.49 sec; Iteration 7051 / 8000) loss: 0.938747
(Time 33.54 sec; Iteration 7061 / 8000) loss: 1.242225
(Time 33.60 sec; Iteration 7071 / 8000) loss: 0.981588
(Time 33.65 sec; Iteration 7081 / 8000) loss: 1.138213
(Time 33.70 sec; Iteration 7091 / 8000) loss: 1.090152
(Time 33.75 sec; Iteration 7101 / 8000) loss: 1.115758
(Time 33.79 sec; Iteration 7111 / 8000) loss: 0.954086
(Time 33.84 sec; Iteration 7121 / 8000) loss: 1.105737
(Time 33.90 sec; Iteration 7131 / 8000) loss: 1.260969
(Time 33.95 sec; Iteration 7141 / 8000) loss: 1.264629
(Time 33.99 sec; Iteration 7151 / 8000) loss: 1.170683
(Time 34.05 sec; Iteration 7161 / 8000) loss: 1.124498
(Time 34.10 sec; Iteration 7171 / 8000) loss: 1.281254
(Time 34.15 sec; Iteration 7181 / 8000) loss: 1.230934
(Time 34.19 sec; Iteration 7191 / 8000) loss: 1.052068
(Epoch 18 / 20) train acc: 0.627000; val_acc: 0.523500
(Time 34.35 sec; Iteration 7201 / 8000) loss: 0.950425
(Time 34.40 sec; Iteration 7211 / 8000) loss: 1.110284
(Time 34.45 sec; Iteration 7221 / 8000) loss: 1.154366
(Time 34.50 sec; Iteration 7231 / 8000) loss: 1.121182
(Time 34.55 sec; Iteration 7241 / 8000) loss: 1.099090
```

```
(Time 34.60 sec; Iteration 7251 / 8000) loss: 1.172849
(Time 34.65 sec; Iteration 7261 / 8000) loss: 1.050562
(Time 34.70 sec; Iteration 7271 / 8000) loss: 1.127165
(Time 34.75 sec; Iteration 7281 / 8000) loss: 0.959455
(Time 34.80 sec; Iteration 7291 / 8000) loss: 1.214903
(Time 34.86 sec; Iteration 7301 / 8000) loss: 1.192749
(Time 34.90 sec; Iteration 7311 / 8000) loss: 1.235171
(Time 34.94 sec; Iteration 7321 / 8000) loss: 1.360503
(Time 34.99 sec; Iteration 7331 / 8000) loss: 1.082100
(Time 35.03 sec; Iteration 7341 / 8000) loss: 1.220841
(Time 35.07 sec; Iteration 7351 / 8000) loss: 1.245445
(Time 35.12 sec; Iteration 7361 / 8000) loss: 1.016539
(Time 35.16 sec; Iteration 7371 / 8000) loss: 1.219668
(Time 35.20 sec; Iteration 7381 / 8000) loss: 1.150572
(Time 35.24 sec; Iteration 7391 / 8000) loss: 1.241113
(Time 35.29 sec; Iteration 7401 / 8000) loss: 1.085364
(Time 35.33 sec; Iteration 7411 / 8000) loss: 1.237737
(Time 35.37 sec; Iteration 7421 / 8000) loss: 1.072891
(Time 35.41 sec; Iteration 7431 / 8000) loss: 1.280416
(Time 35.46 sec; Iteration 7441 / 8000) loss: 1.139609
(Time 35.50 sec; Iteration 7451 / 8000) loss: 1.109688
(Time 35.54 sec; Iteration 7461 / 8000) loss: 1.341762
(Time 35.59 sec; Iteration 7471 / 8000) loss: 0.983673
(Time 35.63 sec; Iteration 7481 / 8000) loss: 1.162268
(Time 35.67 sec; Iteration 7491 / 8000) loss: 1.264371
(Time 35.72 sec; Iteration 7501 / 8000) loss: 1.001572
(Time 35.76 sec; Iteration 7511 / 8000) loss: 0.968682
(Time 35.81 sec; Iteration 7521 / 8000) loss: 1.314246
(Time 35.85 sec; Iteration 7531 / 8000) loss: 1.001908
(Time 35.89 sec; Iteration 7541 / 8000) loss: 1.206905
(Time 35.94 sec; Iteration 7551 / 8000) loss: 1.093572
(Time 35.98 sec; Iteration 7561 / 8000) loss: 1.140373
(Time 36.02 sec; Iteration 7571 / 8000) loss: 1.018711
(Time 36.06 sec; Iteration 7581 / 8000) loss: 1.290428
(Time 36.10 sec; Iteration 7591 / 8000) loss: 1.057528
(Epoch 19 / 20) train acc: 0.602000; val_acc: 0.524000
(Time 36.25 sec; Iteration 7601 / 8000) loss: 1.271423
(Time 36.30 sec; Iteration 7611 / 8000) loss: 1.192346
(Time 36.34 sec; Iteration 7621 / 8000) loss: 1.133291
(Time 36.38 sec; Iteration 7631 / 8000) loss: 1.220565
(Time 36.42 sec; Iteration 7641 / 8000) loss: 1.148030
(Time 36.47 sec; Iteration 7651 / 8000) loss: 1.028671
(Time 36.51 sec; Iteration 7661 / 8000) loss: 1.191155
(Time 36.55 sec; Iteration 7671 / 8000) loss: 1.086887
(Time 36.59 sec; Iteration 7681 / 8000) loss: 1.027545
(Time 36.64 sec; Iteration 7691 / 8000) loss: 1.278987
(Time 36.69 sec; Iteration 7701 / 8000) loss: 1.083637
(Time 36.73 sec; Iteration 7711 / 8000) loss: 1.095324
```

```
(Time 36.77 sec; Iteration 7721 / 8000) loss: 1.052646
(Time 36.82 sec; Iteration 7731 / 8000) loss: 1.109235
(Time 36.86 sec; Iteration 7741 / 8000) loss: 1.181352
(Time 36.90 sec; Iteration 7751 / 8000) loss: 1.024596
(Time 36.95 sec; Iteration 7761 / 8000) loss: 1.122099
(Time 36.99 sec; Iteration 7771 / 8000) loss: 1.157296
(Time 37.04 sec; Iteration 7781 / 8000) loss: 1.123909
(Time 37.08 sec; Iteration 7791 / 8000) loss: 1.217828
(Time 37.12 sec; Iteration 7801 / 8000) loss: 1.031844
(Time 37.16 sec; Iteration 7811 / 8000) loss: 1.185840
(Time 37.21 sec; Iteration 7821 / 8000) loss: 1.149443
(Time 37.25 sec; Iteration 7831 / 8000) loss: 1.080554
(Time 37.29 sec; Iteration 7841 / 8000) loss: 1.062056
(Time 37.34 sec; Iteration 7851 / 8000) loss: 0.950382
(Time 37.38 sec; Iteration 7861 / 8000) loss: 1.127753
(Time 37.42 sec; Iteration 7871 / 8000) loss: 0.933684
(Time 37.46 sec; Iteration 7881 / 8000) loss: 1.178936
(Time 37.51 sec; Iteration 7891 / 8000) loss: 1.198274
(Time 37.55 sec; Iteration 7901 / 8000) loss: 1.156807
(Time 37.59 sec; Iteration 7911 / 8000) loss: 1.114287
(Time 37.63 sec; Iteration 7921 / 8000) loss: 1.234844
(Time 37.68 sec; Iteration 7931 / 8000) loss: 1.126765
(Time 37.72 sec; Iteration 7941 / 8000) loss: 1.230925
(Time 37.77 sec; Iteration 7951 / 8000) loss: 1.185593
(Time 37.81 sec; Iteration 7961 / 8000) loss: 1.087695
(Time 37.85 sec; Iteration 7971 / 8000) loss: 1.145586
(Time 37.89 sec; Iteration 7981 / 8000) loss: 1.047354
(Time 37.94 sec; Iteration 7991 / 8000) loss: 1.010469
(Epoch 20 / 20) train acc: 0.642000; val_acc: 0.522200
load checkpoint file: drive/My Drive/ITE4052/A4/best_two_layer_net.pth
Saved model's accuracy on validation is 0.5327000021934509
```

10 Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in `fully_connected_networks.py`. Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout; we will add this feature soon.

10.1 Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors less than `1e-6`, except for the check on `W1` and `W2` with `reg=0` where your errors should be less than `1e-5`.

[20]:
```
from fully_connected_networks import FullyConnectedNet

reset_seed(0)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = torch.randn(N, D, dtype=torch.float64, device='cuda')
y = torch.randint(C, size=(N,), dtype=torch.int64, device='cuda')

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        reg=reg,
        weight_scale=5e-2,
        dtype=torch.float64,
        device='cuda'
  )

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss.item())

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = ite4052.grad.compute_numeric_gradient(f, model.params[name])
    print('%s relative error: %.2e' % (name, ite4052.grad.rel_error(grad_num,␣
  ↪grads[name])))
```

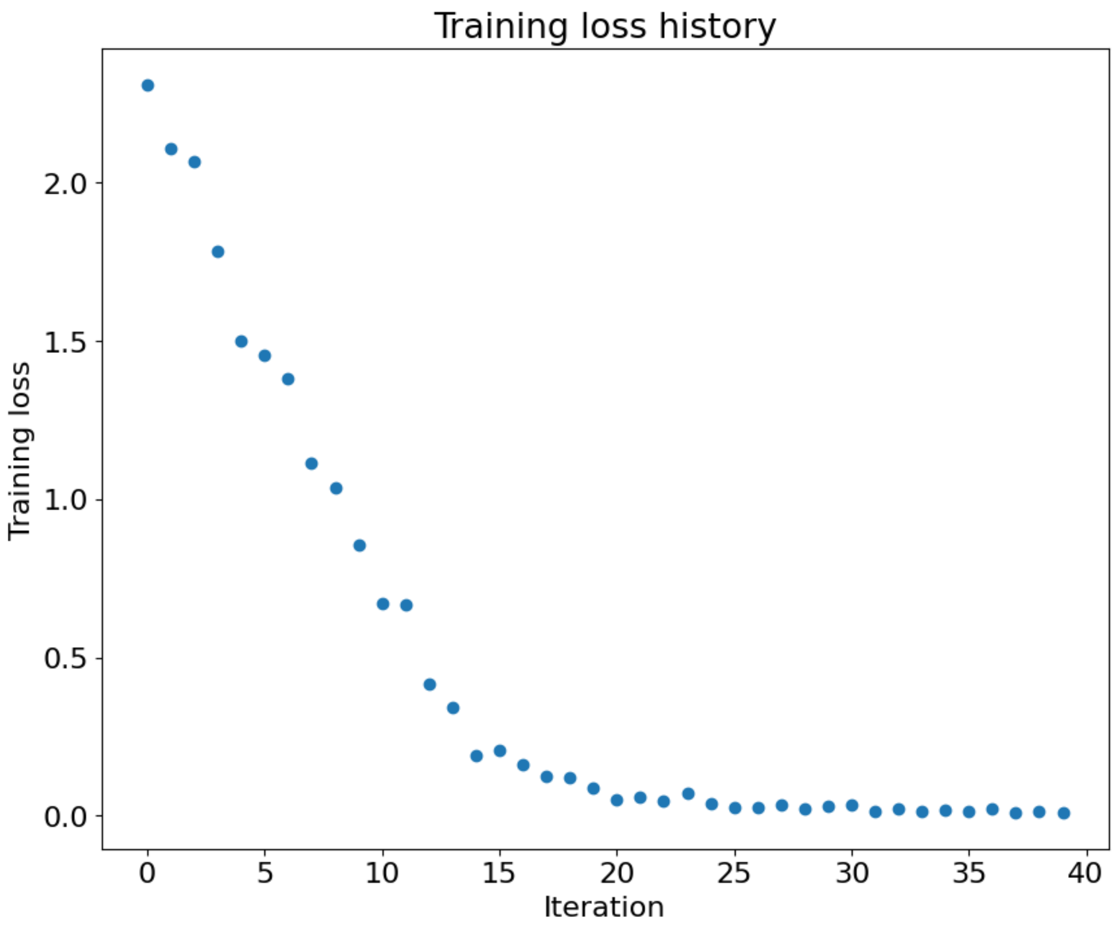
```
Running check with reg =  0
Initial loss:  2.3053575717037686
W1 relative error: 6.06e-08
W2 relative error: 1.02e-07
W3 relative error: 5.89e-08
b1 relative error: 1.28e-07
b2 relative error: 2.05e-08
b3 relative error: 3.41e-09
Running check with reg =  3.14
Initial loss:  12.278358041494133
W1 relative error: 5.60e-09
W2 relative error: 8.54e-09
W3 relative error: 1.25e-08
b1 relative error: 5.76e-07
b2 relative error: 1.46e-07
b3 relative error: 1.53e-08
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy within 20 epochs.

[21]:
```
from fully_connected_networks import FullyConnectedNet,␣
 ↪get_three_layer_network_params

# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.
reset_seed(0)

num_train = 50
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}

# Update parameters in get_three_layer_network_params
weight_scale, learning_rate = get_three_layer_network_params()

model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=torch.float32, device='cuda')
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                optim_config={
                  'learning_rate': learning_rate,
                },
                device='cuda',
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Time 0.03 sec; Iteration 1 / 40) loss: 2.310387
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.102700
(Epoch 1 / 20) train acc: 0.400000; val_acc: 0.139600
(Epoch 2 / 20) train acc: 0.560000; val_acc: 0.175600
(Epoch 3 / 20) train acc: 0.680000; val_acc: 0.171700
(Epoch 4 / 20) train acc: 0.860000; val_acc: 0.195100
(Epoch 5 / 20) train acc: 0.820000; val_acc: 0.194700
(Time 0.17 sec; Iteration 11 / 40) loss: 0.672492
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.188300
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.193100
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.192400
```

```
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.192000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.199900
(Time 0.29 sec; Iteration 21 / 40) loss: 0.051000
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.200100
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.194400
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.196100
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.195000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.202600
(Time 0.40 sec; Iteration 31 / 40) loss: 0.035407
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.197400
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.199100
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.199500
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.197500
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.198900
```

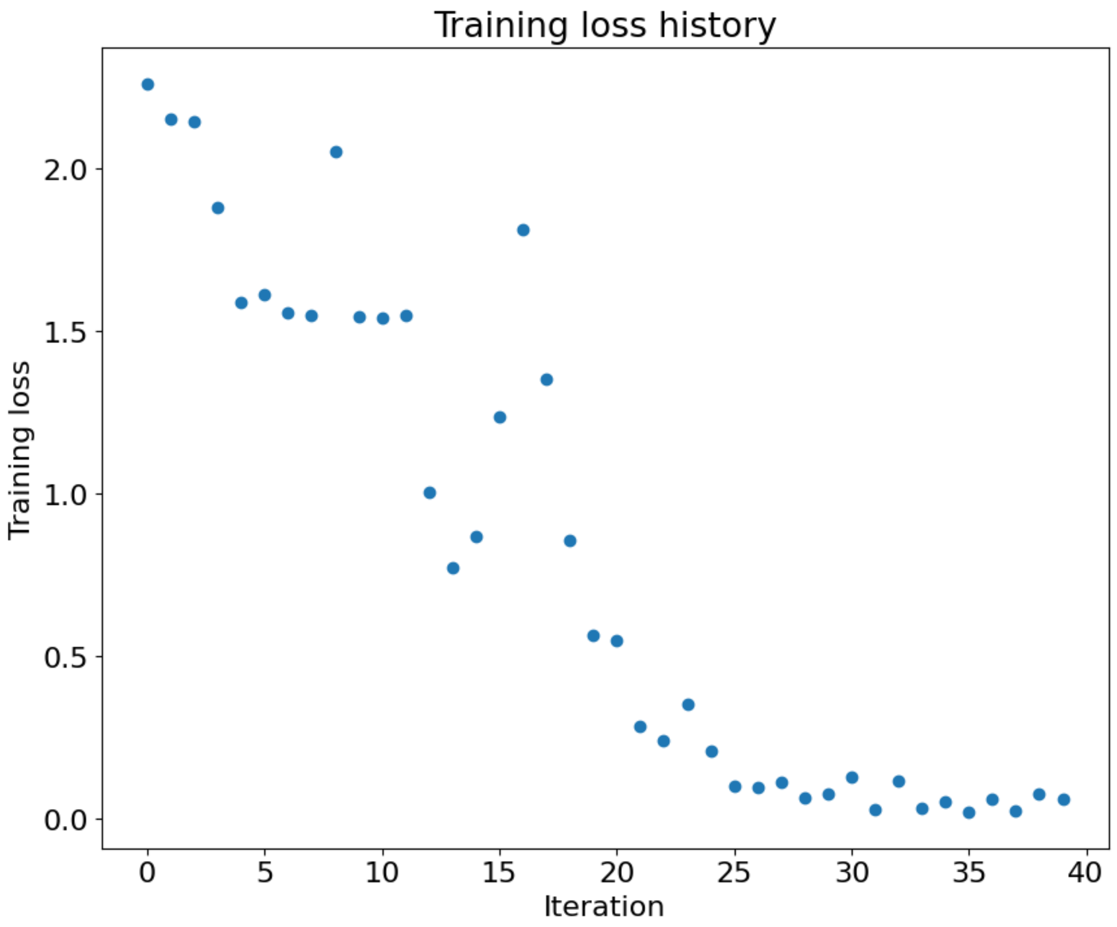


Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

```
[22]: from fully_connected_networks import FullyConnectedNet,␣
       ↪get_five_layer_network_params

      # TODO: Use a five-layer Net to overfit 50 training examples by
      # tweaking just the learning rate and initialization scale.
      reset_seed(0)

      num_train = 50
      small_data = {
        'X_train': data_dict['X_train'][:num_train],
        'y_train': data_dict['y_train'][:num_train],
        'X_val': data_dict['X_val'],
        'y_val': data_dict['y_val'],
      }


      # Update parameters in get_three_layer_network_params
      weight_scale, learning_rate = get_five_layer_network_params()

      # Run models and solver with parameters
      model = FullyConnectedNet([100, 100, 100, 100],
                      weight_scale=weight_scale, dtype=torch.float32, device='cuda')
      solver = Solver(model, small_data,
                      print_every=10, num_epochs=20, batch_size=25,
                      optim_config={
                        'learning_rate': learning_rate,
                      },
                      device='cuda',
               )
      # Turn off keep_best_params to allow final weights to be saved, instead of best␣
       ↪weights on validation set.
      solver.train(return_best_params=False)

      plt.plot(solver.loss_history, 'o')
      plt.title('Training loss history')
      plt.xlabel('Iteration')
      plt.ylabel('Training loss')
      plt.show()
```

```
(Time 0.00 sec; Iteration 1 / 40) loss: 2.262105
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.101600
(Epoch 1 / 20) train acc: 0.440000; val_acc: 0.127000
(Epoch 2 / 20) train acc: 0.480000; val_acc: 0.123100
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.117200
(Epoch 4 / 20) train acc: 0.200000; val_acc: 0.101700
(Epoch 5 / 20) train acc: 0.540000; val_acc: 0.155500
(Time 0.32 sec; Iteration 11 / 40) loss: 1.542219
```

```
(Epoch 6 / 20) train acc: 0.780000; val_acc: 0.178500
(Epoch 7 / 20) train acc: 0.760000; val_acc: 0.169100
(Epoch 8 / 20) train acc: 0.460000; val_acc: 0.125100
(Epoch 9 / 20) train acc: 0.840000; val_acc: 0.179100
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.185400
(Time 0.56 sec; Iteration 21 / 40) loss: 0.548560
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.190600
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.184100
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.188900
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.190500
(Epoch 15 / 20) train acc: 0.940000; val_acc: 0.187400
(Time 0.79 sec; Iteration 31 / 40) loss: 0.129615
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.184800
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.197600
(Epoch 18 / 20) train acc: 0.980000; val_acc: 0.200000
(Epoch 19 / 20) train acc: 0.980000; val_acc: 0.187500
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.192600
```



If you're satisfied with your model's performance, save the overfit model. Just a sanity check, we evaluate it one the training set again to verify that the saved weights have the correct performance.

[23]:
```python
# Set path
path = os.path.join(GOOGLE_DRIVE_PATH, 'best_overfit_five_layer_net.pth')
solver.model.save(path)


# Create a new instance  -- Note that hidden dims being different doesn't␣
 ↪matter here.
model = FullyConnectedNet(hidden_dims=[100, ], dtype=torch.float32,␣
 ↪device='cuda')
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                optim_config={
                  'learning_rate': learning_rate,
                },
                device='cuda',
         )


# Load model
solver.model.load(path, dtype=torch.float32, device='cuda')

# Evaluate on validation set
accuracy = solver.check_accuracy(solver.X_train, solver.y_train)
print(f"Saved model's accuracy on small train is {accuracy}")
```

```
Saved in drive/My Drive/ITE4052/A4/best_overfit_five_layer_net.pth
load checkpoint file: drive/My Drive/ITE4052/A4/best_overfit_five_layer_net.pth
Saved model's accuracy on small train is 1.0
```

11 Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

11.1 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

We will implement 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 tensor giving the current weights.
  - dw: A tensor 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.
```

We provide the implementation of the SGD update rule for your reference in `fully_connected_networks.py`

Now **implement** the SGD+Momentum update rule using the same interface. Run the following to check your implementation of SGD+Momentum. You should see errors less than `1e-7`.

[24]:
```
from fully_connected_networks import sgd_momentum

reset_seed(0)

N, D = 4, 5
w = torch.linspace(-0.4, 0.6, steps=N*D, dtype=torch.float64, device='cuda').
 ↪reshape(N, D)
dw = torch.linspace(-0.6, 0.4, steps=N*D, dtype=torch.float64, device='cuda').
 ↪reshape(N, D)
v = torch.linspace(0.6, 0.9, steps=N*D, dtype=torch.float64, device='cuda').
 ↪reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = torch.tensor([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]],
   dtype=torch.float64, device='cuda')
expected_velocity = torch.tensor([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]],
  dtype=torch.float64, device='cuda')

# Should see relative errors around e-8 or less
print('next_w error: ', ite4052.grad.rel_error(next_w, expected_next_w))
print('velocity error: ', ite4052.grad.rel_error(expected_velocity,␣
 ↪config['velocity']))
```

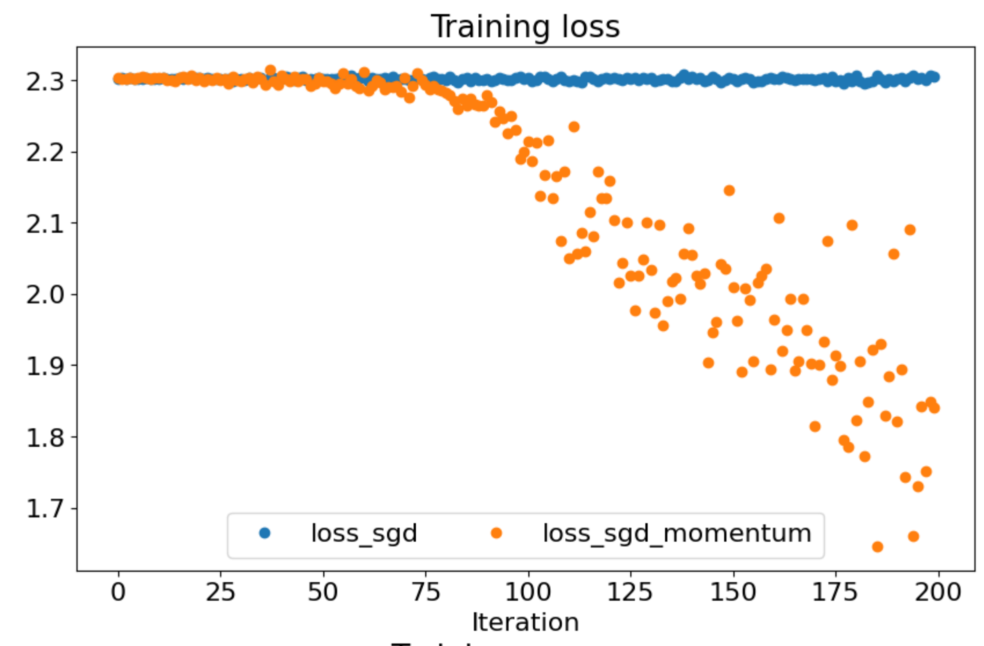
```
next_w error:  1.6802078709310813e-09
velocity error:  2.9254212825785614e-09
```

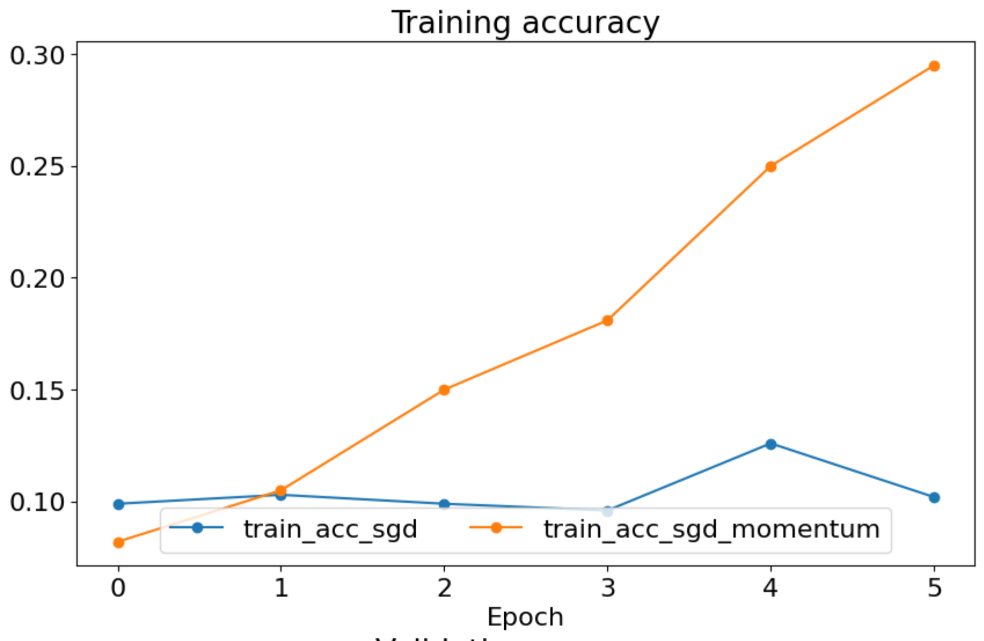
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

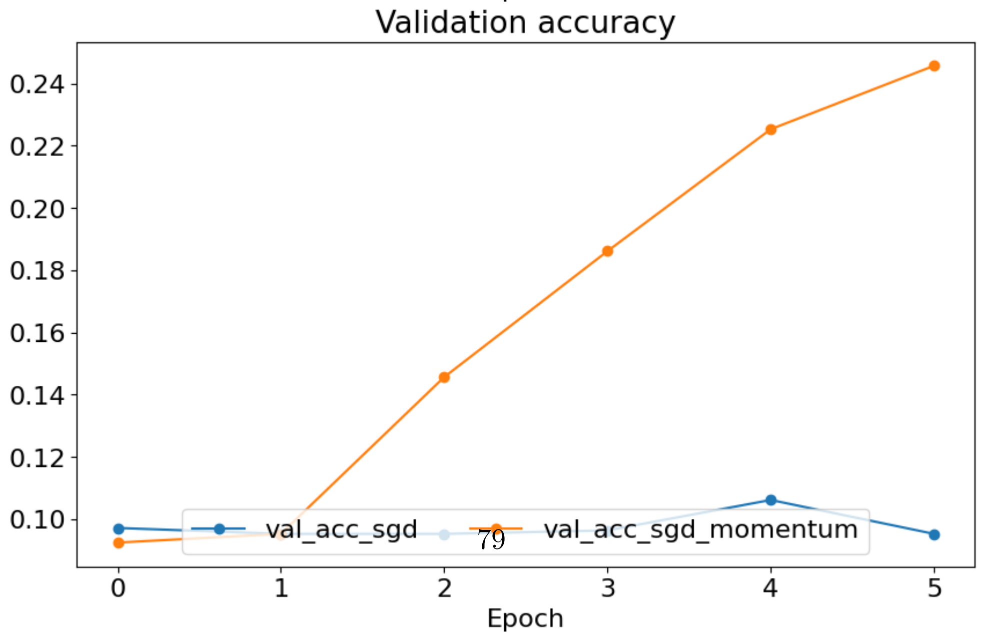
[25]:
```
from fully_connected_networks import FullyConnectedNet, sgd, sgd_momentum

# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.
reset_seed(0)

num_train = 4000
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}

solvers = {}

for update_rule_name, update_rule_fn in [('sgd', sgd), ('sgd_momentum',␣
 ↪sgd_momentum)]:
  print('running with ', update_rule_name)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2,
                            dtype=torch.float32, device='cuda')

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule_fn,
                  optim_config={
                    'learning_rate': 5e-2,
                  },
                  print_every=1000,
                  verbose=True,
                  device='cuda')
  solvers[update_rule_name] = solver
  solver.train()
  print()
```

```
plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')
for update_rule, solver in solvers.items():
  plt.plot(solver.loss_history, 'o', label="loss_%s" % update_rule)
plt.legend(loc='lower center', ncol=4)

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')
for update_rule, solver in solvers.items():
  plt.plot(solver.train_acc_history, '-o', label="train_acc_%s" % update_rule)
plt.legend(loc='lower center', ncol=4)


plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')
for update_rule, solver in solvers.items():
  plt.plot(solver.val_acc_history, '-o', label="val_acc_%s" % update_rule)
plt.legend(loc='lower center', ncol=4)

plt.gcf().set_size_inches(10, 20)
plt.show()
```

```
running with  sgd
(Time 0.01 sec; Iteration 1 / 200) loss: 2.302603
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.097100
(Epoch 1 / 5) train acc: 0.103000; val_acc: 0.095200
(Epoch 2 / 5) train acc: 0.099000; val_acc: 0.095200
(Epoch 3 / 5) train acc: 0.096000; val_acc: 0.096300
(Epoch 4 / 5) train acc: 0.126000; val_acc: 0.106100
(Epoch 5 / 5) train acc: 0.102000; val_acc: 0.095200

running with  sgd_momentum
(Time 0.00 sec; Iteration 1 / 200) loss: 2.303541
(Epoch 0 / 5) train acc: 0.082000; val_acc: 0.092400
(Epoch 1 / 5) train acc: 0.105000; val_acc: 0.095200
(Epoch 2 / 5) train acc: 0.150000; val_acc: 0.145600
(Epoch 3 / 5) train acc: 0.181000; val_acc: 0.186100
(Epoch 4 / 5) train acc: 0.250000; val_acc: 0.225300
(Epoch 5 / 5) train acc: 0.295000; val_acc: 0.245700
```


Training loss
2.3
2.2
2.1
2.0
1.9
1.8
1.7
loss_sgd
loss_sgd_momentum
0
25
50
75
100
125
150
175
200
Iteration


Training accuracy
0.30
0.25
0.20
0.15
0.10
train_acc_sgd
train_acc_sgd_momentum
0
1
2
3
4
5
Epoch


Validation accuracy
0.24
0.22
0.20
0.18
0.16
0.14
0.12
0.10
val_acc_sgd
val_acc_sgd_momentum
0
1
2
3
4
5
Epoch

11.2 RMSProp

RMSProp [1] is an update rule that set per-parameter learning rates by using a running average of the second moments of gradients.

Implement the RMSProp update rule in the `rmsprop` function in `fully_connected_networks.py`. Run the following to test your RMSProp implementation. You should see errors less than `1e-6`.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[26]:
```
# Test RMSProp implementation
from fully_connected_networks import rmsprop

reset_seed(0)

N, D = 4, 5
w = torch.linspace(-0.4, 0.6, steps=N*D, dtype=torch.float64, device='cuda').
 ↪reshape(N, D)
dw = torch.linspace(-0.6, 0.4, steps=N*D, dtype=torch.float64, device='cuda').
 ↪reshape(N, D)
cache = torch.linspace(0.6, 0.9, steps=N*D, dtype=torch.float64, device='cuda').
 ↪reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = torch.tensor([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]],
   dtype=torch.float64, device='cuda')
expected_cache = torch.tensor([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]],
   dtype=torch.float64, device='cuda')

print('next_w error: ', ite4052.grad.rel_error(expected_next_w, next_w))
print('cache error: ', ite4052.grad.rel_error(expected_cache, config['cache']))
```

```
next_w error:  4.064797880829826e-09
cache error:  1.8620321382570356e-09
```

11.3 Adam

Adam [2] extends RMSprop with a first-order gradient cache similar to momentum, and a bias correction mechanism to prevent large steps at the start of optimization. Adam is one of the most commonly used update rules used in practice for training deep neural networks.

Implement the Adam update rule in the `adam` function in `fully_connected_networks.py`. Run the following to test your Adam implementation. You should see error less than `1e-6` for `next_w`, and errors less than `1e-8` for `v` and `m`:

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

[27]:
```
# Test Adam implementation
from fully_connected_networks import adam

reset_seed(0)

N, D = 4, 5
w = torch.linspace(-0.4, 0.6, steps=N*D, dtype=torch.float64, device='cuda').
  ↪reshape(N, D)
dw = torch.linspace(-0.6, 0.4, steps=N*D, dtype=torch.float64, device='cuda').
  ↪reshape(N, D)
m = torch.linspace(0.6, 0.9, steps=N*D, dtype=torch.float64, device='cuda').
  ↪reshape(N, D)
v = torch.linspace(0.7, 0.5, steps=N*D, dtype=torch.float64, device='cuda').
  ↪reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = torch.tensor([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]],
   dtype=torch.float64, device='cuda')
expected_v = torch.tensor([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]],
   dtype=torch.float64, device='cuda')
expected_m = torch.tensor([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
 [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]],
  dtype=torch.float64, device='cuda')

# You should see relative errors around e-7 or less
print('next_w error: ', ite4052.grad.rel_error(expected_next_w, next_w))
print('v error: ', ite4052.grad.rel_error(expected_v, config['v']))
print('m error: ', ite4052.grad.rel_error(expected_m, config['m']))
```

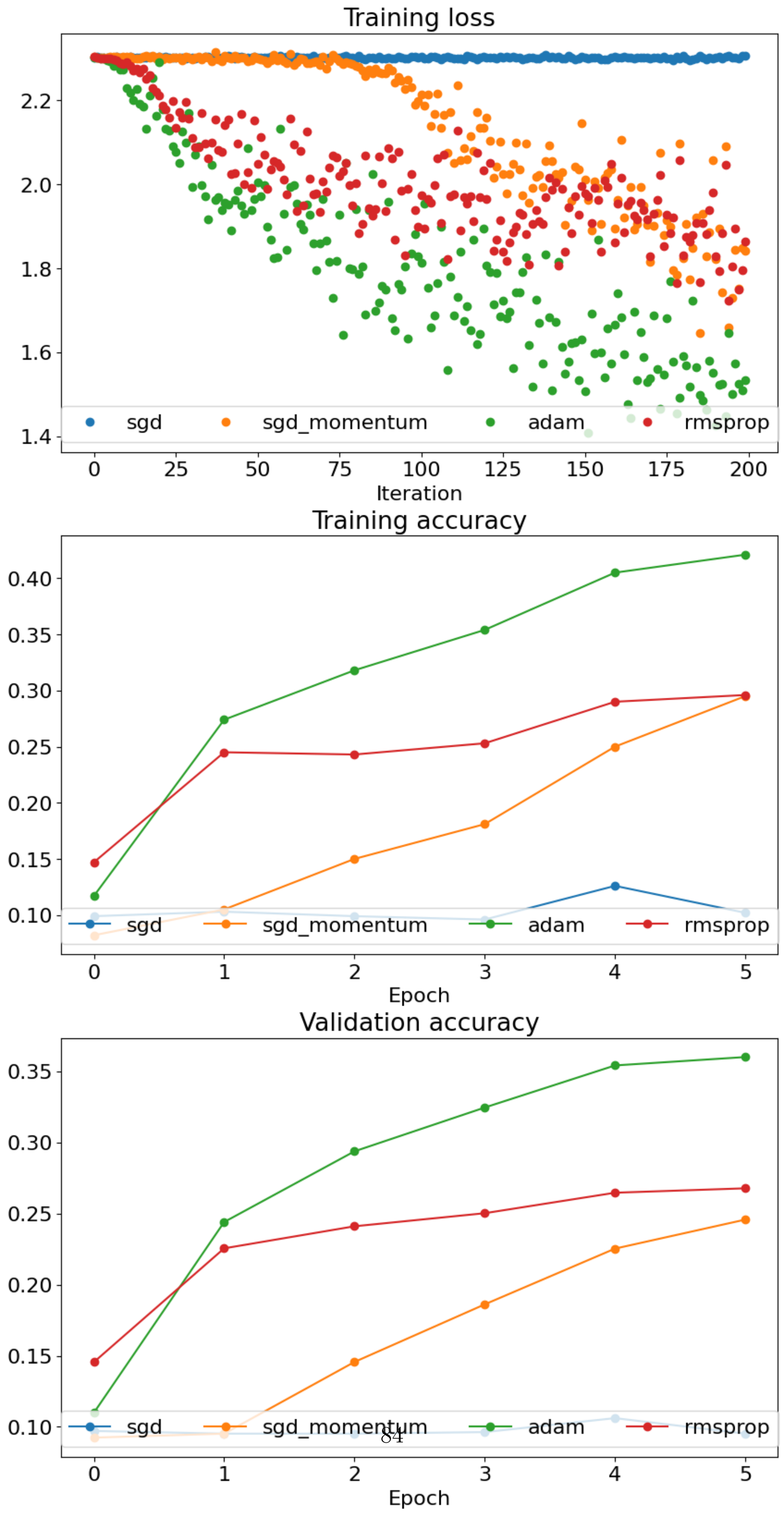
```
next_w error:  3.756728297598868e-09
v error:  3.4048987160545265e-09
m error:  2.786377729853651e-09
```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[28]:
```
# Test Adam implementation
from fully_connected_networks import adam, rmsprop, FullyConnectedNet

for update_rule_name, update_rule_fn, learning_rate in [('adam', adam, 1e-3),␣
↪('rmsprop', rmsprop, 1e-4)]:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2,␣
↪device='cuda')

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule_fn,
                  optim_config={
                    'learning_rate': learning_rate
                  },
                  print_every=1000,
                  verbose=True, device='cuda')
  solvers[update_rule_name] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')
for update_rule, solver in list(solvers.items()):
  plt.plot(solver.loss_history, 'o', label=update_rule)
plt.legend(loc='lower center', ncol=4)

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')
for update_rule, solver in list(solvers.items()):
  plt.plot(solver.train_acc_history, '-o', label=update_rule)
```

```python
plt.legend(loc='lower center', ncol=4)

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')
for update_rule, solver in list(solvers.items()):
  plt.plot(solver.val_acc_history, '-o', label=update_rule)
plt.legend(loc='lower center', ncol=4)

plt.gcf().set_size_inches(10, 20)
plt.show()
```

```
running with  sgd_momentum
(Time 0.00 sec; Iteration 1 / 200) loss: 2.302603
(Epoch 0 / 5) train acc: 0.117000; val_acc: 0.109900
(Epoch 1 / 5) train acc: 0.274000; val_acc: 0.244000
(Epoch 2 / 5) train acc: 0.318000; val_acc: 0.293700
(Epoch 3 / 5) train acc: 0.354000; val_acc: 0.324500
(Epoch 4 / 5) train acc: 0.405000; val_acc: 0.354100
(Epoch 5 / 5) train acc: 0.421000; val_acc: 0.360000

running with  sgd_momentum
(Time 0.00 sec; Iteration 1 / 200) loss: 2.303541
(Epoch 0 / 5) train acc: 0.147000; val_acc: 0.145500
(Epoch 1 / 5) train acc: 0.245000; val_acc: 0.225500
(Epoch 2 / 5) train acc: 0.243000; val_acc: 0.241000
(Epoch 3 / 5) train acc: 0.253000; val_acc: 0.250200
(Epoch 4 / 5) train acc: 0.290000; val_acc: 0.264600
(Epoch 5 / 5) train acc: 0.296000; val_acc: 0.267700
```


Training loss
Iteration
sgd
sgd_momentum
adam
rmsprop
Training accuracy
Epoch
Validation accuracy
Epoch

12 Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some output activations to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

12.1 Dropout: forward

Implement the forward pass for dropout in `fully_connected_networks.py`. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Run the following to test your dropout implementation. The mean of the output should be approximately the same during training and testing. During training the number of outputs set to zero should be approximately equal to the drop probability `p`, and during testing no outputs should be set to zero.

[29]:
```
from fully_connected_networks import Dropout

reset_seed(0)
x = torch.randn(500, 500, dtype=torch.float64, device='cuda') + 10

for p in [0.25, 0.4, 0.7]:
  out, _ = Dropout.forward(x, {'mode': 'train', 'p': p})
  out_test, _ = Dropout.forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean().item())
  print('Mean of train-time output: ', out.mean().item())
  print('Mean of test-time output: ', out_test.mean().item())
  print('Fraction of train-time output set to zero: ', (out == 0).type(torch.
↪float32).mean().item())
  print('Fraction of test-time output set to zero: ', (out_test == 0).
↪type(torch.float32).mean().item())
  print()
```

```
Running tests with p =  0.25
Mean of input:  9.997330335850453
Mean of train-time output:  7.492388669496627
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.2505599856376648
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
```

```
Mean of input:  9.997330335850453
Mean of train-time output:  5.984183896026179
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.40133199095726013
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  9.997330335850453
Mean of train-time output:  3.0022526715624593
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.6997119784355164
Fraction of test-time output set to zero:  0.0
```

12.2 Dropout: backward

Implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

[30]:
```python
from fully_connected_networks import Dropout

reset_seed(0)
x = torch.randn(10, 10, dtype=torch.float64, device='cuda') + 10
dout = torch.randn_like(x)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 0}
out, cache = Dropout.forward(x, dropout_param)
dx = Dropout.backward(dout, cache)
dx_num = ite4052.grad.compute_numeric_gradient(lambda xx: Dropout.forward(xx,␣
    ↪dropout_param)[0], x, dout)

# Error should be around e-10 or less
print('dx relative error: ', ite4052.grad.rel_error(dx, dx_num))
```

```
dx relative error:  3.038735480167295e-09
```

13 Fully-connected nets with dropout

Modify your implementation of `FullyConnectedNet` to use dropout. Specifically, if the constructor of the network receives a value that is not 0 for the `dropout` parameter, then the net should add a dropout layer immediately after every ReLU nonlinearity.

After doing so, run the following to numerically gradient-check your implementation. You should see errors less than `1e-5`, and different dropout rates should result different error values.

[31]:
```python
from fully_connected_networks import FullyConnectedNet

reset_seed(0)
```

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = torch.randn(N, D, dtype=torch.float64, device='cuda')
y = torch.randint(C, size=(N,), dtype=torch.int64, device='cuda')

for dropout in [0, 0.25, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dropout=dropout,
                            seed=0, dtype=torch.float64, device='cuda')

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss.item())

  # Relative errors should be around e-5 or less.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = ite4052.grad.compute_numeric_gradient(f, model.params[name])
    print('%s relative error: %.2e' % (name, ite4052.grad.rel_error(grad_num,␣
↪grads[name])))
  print()
```

```
Running check with dropout =  0
Initial loss:  2.3053575717037686
W1 relative error: 6.06e-08
W2 relative error: 1.02e-07
W3 relative error: 5.89e-08
b1 relative error: 1.28e-07
b2 relative error: 2.05e-08
b3 relative error: 3.41e-09

Running check with dropout =  0.25
Initial loss:  2.3036981550386986
W1 relative error: 6.51e-08
W2 relative error: 9.48e-08
W3 relative error: 6.85e-08
b1 relative error: 1.78e-07
b2 relative error: 2.10e-08
b3 relative error: 3.49e-09

Running check with dropout =  0.5
Initial loss:  2.2980030684524655
W1 relative error: 4.60e-08
W2 relative error: 7.66e-08
W3 relative error: 5.05e-08
b1 relative error: 7.37e-08
b2 relative error: 3.19e-08
```

```
b3 relative error: 2.82e-09
```

13.1 Regularization experiment

To get a sense of the way that dropout can regularize a neural network, we will train three different two-layer networks:

1. Hidden size 256, dropout = 0
2. Hidden size 512, dropout = 0
3. Hidden size 512, dropout = 0.5

We will then visualize the training and validation accuracies of these three networks.

[32]:
```
from fully_connected_networks import FullyConnectedNet

# Train two identical nets, one with dropout and one without
reset_seed(0)
num_train = 20000
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}

solvers = {}
dropout_choices = [0, 0, 0.5]
width_choices = [256, 512, 512]
for dropout, width in zip(dropout_choices, width_choices):
# for dropout in dropout_choices:
  model = FullyConnectedNet([width], dropout=dropout, dtype=torch.float32,␣
↪device='cuda')
  print('Training a model with dropout=%.2f and width=%d' % (dropout, width))

  solver = Solver(model, small_data,
                  num_epochs=100, batch_size=512,
                  update_rule=adam,
                  optim_config={
                    'learning_rate': 5e-3,
                  },
                  print_every=100000, print_acc_every=10,
                  verbose=True, device='cuda')
  solver.train()
  solvers[(dropout, width)] = solver
  print()
```

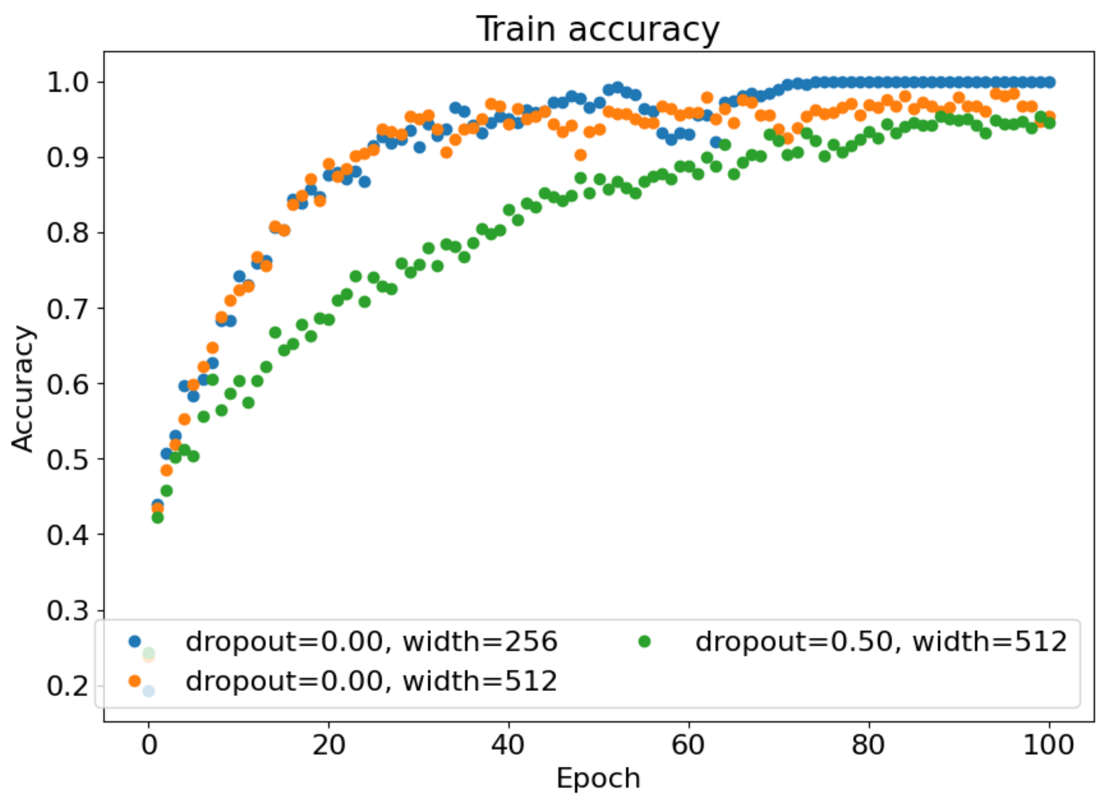
```
Training a model with dropout=0.00 and width=256
(Time 0.00 sec; Iteration 1 / 3900) loss: 2.304467
```

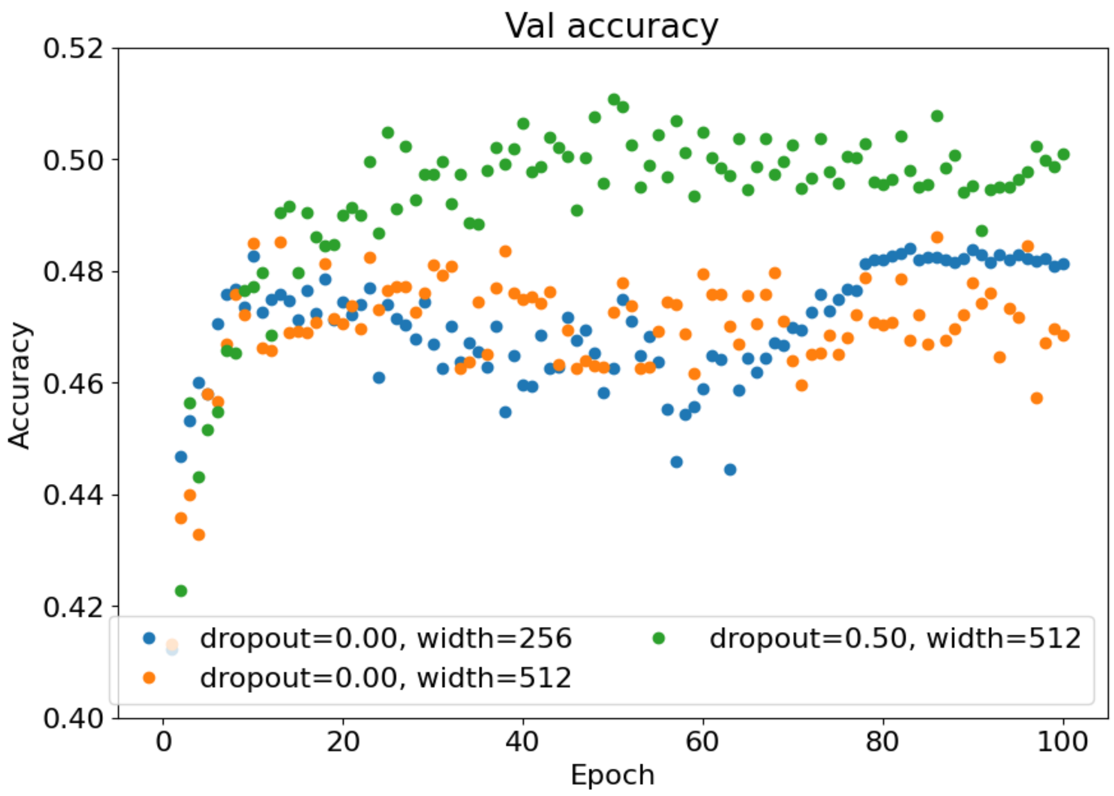
```
(Epoch 0 / 100) train acc: 0.193000; val_acc: 0.198200
(Epoch 10 / 100) train acc: 0.742000; val_acc: 0.482600
(Epoch 20 / 100) train acc: 0.876000; val_acc: 0.474400
(Epoch 30 / 100) train acc: 0.913000; val_acc: 0.467000
(Epoch 40 / 100) train acc: 0.951000; val_acc: 0.459700
(Epoch 50 / 100) train acc: 0.973000; val_acc: 0.462600
(Epoch 60 / 100) train acc: 0.930000; val_acc: 0.458900
(Epoch 70 / 100) train acc: 0.989000; val_acc: 0.469900
(Epoch 80 / 100) train acc: 1.000000; val_acc: 0.481900
(Epoch 90 / 100) train acc: 1.000000; val_acc: 0.483900
(Epoch 100 / 100) train acc: 1.000000; val_acc: 0.481300

Training a model with dropout=0.00 and width=512
(Time 0.00 sec; Iteration 1 / 3900) loss: 2.302387
(Epoch 0 / 100) train acc: 0.239000; val_acc: 0.220000
(Epoch 10 / 100) train acc: 0.723000; val_acc: 0.484900
(Epoch 20 / 100) train acc: 0.891000; val_acc: 0.470500
(Epoch 30 / 100) train acc: 0.951000; val_acc: 0.481100
(Epoch 40 / 100) train acc: 0.944000; val_acc: 0.475000
(Epoch 50 / 100) train acc: 0.937000; val_acc: 0.472700
(Epoch 60 / 100) train acc: 0.958000; val_acc: 0.479500
(Epoch 70 / 100) train acc: 0.937000; val_acc: 0.463900
(Epoch 80 / 100) train acc: 0.969000; val_acc: 0.470400
(Epoch 90 / 100) train acc: 0.979000; val_acc: 0.477900
(Epoch 100 / 100) train acc: 0.954000; val_acc: 0.468400

Training a model with dropout=0.50 and width=512
(Time 0.01 sec; Iteration 1 / 3900) loss: 2.303460
(Epoch 0 / 100) train acc: 0.244000; val_acc: 0.237300
(Epoch 10 / 100) train acc: 0.604000; val_acc: 0.477100
(Epoch 20 / 100) train acc: 0.685000; val_acc: 0.490000
(Epoch 30 / 100) train acc: 0.758000; val_acc: 0.497400
(Epoch 40 / 100) train acc: 0.831000; val_acc: 0.506400
(Epoch 50 / 100) train acc: 0.871000; val_acc: 0.510700
(Epoch 60 / 100) train acc: 0.887000; val_acc: 0.504900
(Epoch 70 / 100) train acc: 0.922000; val_acc: 0.502500
(Epoch 80 / 100) train acc: 0.933000; val_acc: 0.495400
(Epoch 90 / 100) train acc: 0.948000; val_acc: 0.495200
(Epoch 100 / 100) train acc: 0.946000; val_acc: 0.501000
```

If everything worked as expected, you should see that the network with dropout has lower training accuracies than the networks without dropout, but that it achieves higher validation accuracies.

You should also see that a network with width 512 and dropout 0.5 achieves higher validation accuracies than a network with width 256 and no dropout. This demonstrates that reducing the model size is not generally an effective regularization strategy – it's often better to use a larger model with explicit regularization.

```
[33]: plt.subplot(2, 1, 1)
for (dropout, width), solver in solvers.items():
  train_acc = solver.train_acc_history
  label = 'dropout=%.2f, width=%d' % (dropout, width)
  plt.plot(train_acc, 'o', label=label)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(2, 1, 2)
for (dropout, width), solver in solvers.items():
  val_acc = solver.val_acc_history
  label = 'dropout=%.2f, width=%d' % (dropout, width)
  plt.plot(val_acc, 'o', label=label)
plt.ylim(0.4, 0.52)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(10, 15)
plt.show()
```


Train accuracy
Accuracy
Epoch
dropout=0.00, width=256
dropout=0.00, width=512
dropout=0.50, width=512


Val accuracy
Accuracy
Epoch
dropout=0.00, width=256
dropout=0.00, width=512
dropout=0.50, width=512

convolutional_networks

May 15, 2024

1 ITE4052 Assignment 4-2: Convolutional Neural Networks and Batch Normalization

- This material draws from EECS 498-007/598-005 (Justin Johnson)

Before we start, please put your name and HYID in following format Firstname LASTNAME, #00000000 // e.g.) Sukmin Yun, #12345678

Your Answer:
Junwoo Park, #2021006253

1.1 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment, same as Assignment 1. You'll need to rerun this setup code each time you start the notebook.

First, run this cell load the autoreload extension. This allows us to edit .py source files, and re-import them into the notebook for a seamless editing and debugging experience.

[77]:
```
%load_ext autoreload
%autoreload 2
```

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

1.1.1 Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

Now recall the path in your Google Drive where you uploaded this notebook, fill it in below. If everything is working correctly then running the folowing cell should print the filenames from the assignment:

```
['convolutional_networks.ipynb', 'fully_connected_networks.ipynb', 'ite4052', 'convolutional_ne
```

```
[ ]: import os

     # TODO: Fill in the Google Drive path where you uploaded the assignment
     # Example: If you create a ITE4052 folder and put all the files under A4␣
      ↪folder, then 'ITE4052/A4'
     GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'ITE4052/A4'
     GOOGLE_DRIVE_PATH = os.path.join('drive', 'My Drive',␣
      ↪GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
     print(os.listdir(GOOGLE_DRIVE_PATH))
```

```
['makepdf.py', 'collectSubmission.sh', 'collect_submission.ipynb',
'a4_helper.py', 'ite4052', '__pycache__', 'overfit_deepconvnet.pth',
'one_minute_deepconvnet.pth', 'convolutional_networks.py',
'fully_connected_networks.py', 'best_two_layer_net.pth',
'best_overfit_five_layer_net.pth', 'fully_connected_networks.ipynb',
'convolutional_networks.ipynb']
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run th following cell to allow us to import from the `.py` files of this assignment. If it works correctly, it should print the message:

```
Hello from convolutional_networks.py!
Hello from a4_helper.py!
```

as well as the last edit time for the file `convolutional_networks.py`.

```
[ ]: import sys
     sys.path.append(GOOGLE_DRIVE_PATH)

     import time, os
     os.environ["TZ"] = "US/Eastern"
     time.tzset()

     from convolutional_networks import hello_convolutional_networks
     hello_convolutional_networks()

     from a4_helper import hello_helper
     hello_helper()

     convolutional_networks_path = os.path.join(GOOGLE_DRIVE_PATH,␣
      ↪'convolutional_networks.py')
     convolutional_networks_edit_time = time.ctime(os.path.
      ↪getmtime(convolutional_networks_path))
```

```
print('convolutional_networks.py last edited on %s' %␣
 ↪convolutional_networks_edit_time)
```

```
Hello from convolutional_networks.py!
Hello from a4_helper.py!
convolutional_networks.py last edited on Wed May 15 11:36:46 2024
```

2 Data preprocessing

2.1 Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

[ ]:
```
import ite4052
import torch
import torchvision
import matplotlib.pyplot as plt
import statistics
import random
import time
import math
%matplotlib inline

from ite4052 import reset_seed, Solver

plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['font.size'] = 16
```

Starting in this assignment, we will use the GPU to accelerate our computation. Run this cell to make sure you are using a GPU.

[ ]:
```
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

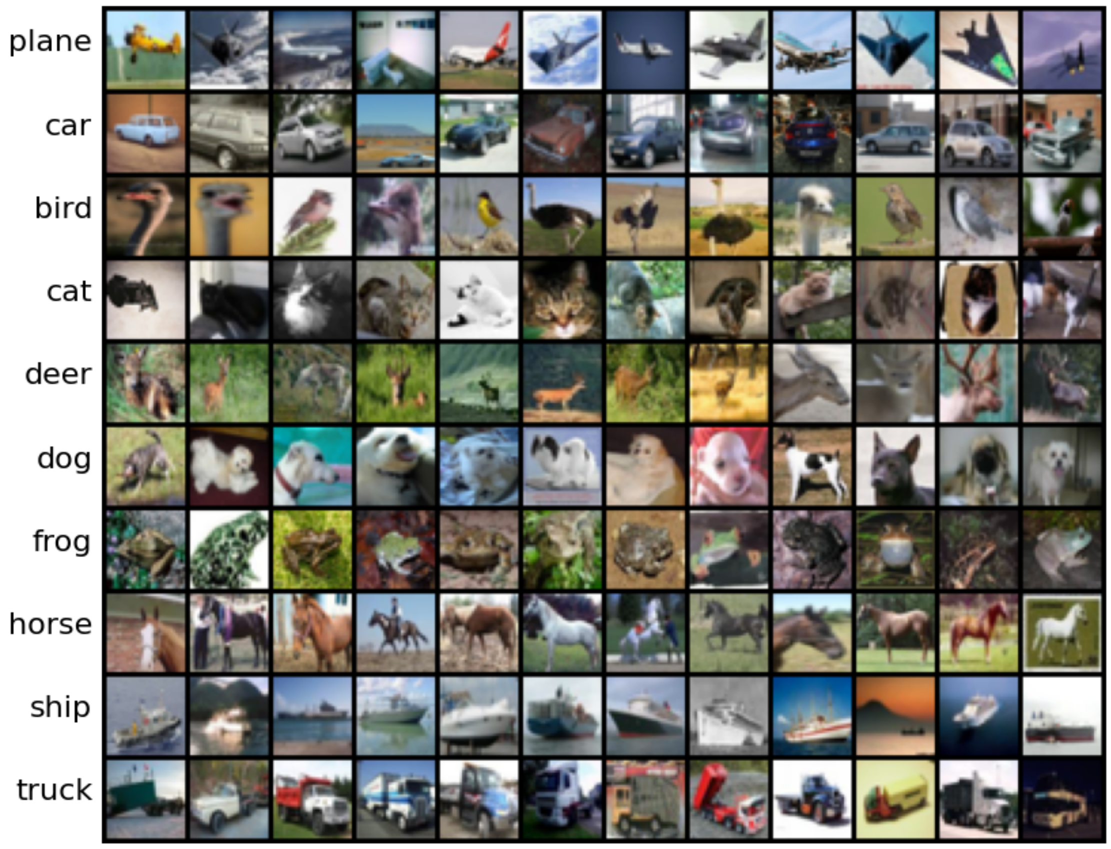
2.2 Load the CIFAR-10 dataset

Then, we will first load the CIFAR-10 dataset, same as knn. The utility function `get_CIFAR10_data()` in `helper_functions` returns the entire CIFAR-10 dataset as a set of six **Torch tensors** while also preprocessing the RGB images:

- `X_train` contains all training images (real numbers in the range $[0, 1]$)
- `y_train` contains all training labels (integers in the range $[0, 9]$)
- `X_val` contains all validation images
- `y_val` contains all validation labels

- `X_test` contains all test images
- `y_test` contains all test labels

```
[ ]: # Invoke the above function to get our data.
import ite4052

ite4052.reset_seed(0)
data_dict = ite4052.data.preprocess_cifar10(cuda=True, dtype=torch.float64,␣
 ↪flatten=False)
print('Train data shape: ', data_dict['X_train'].shape)
print('Train labels shape: ', data_dict['y_train'].shape)
print('Validation data shape: ', data_dict['X_val'].shape)
print('Validation labels shape: ', data_dict['y_val'].shape)
print('Test data shape: ', data_dict['X_test'].shape)
print('Test labels shape: ', data_dict['y_test'].shape)
```



```
Train data shape:  torch.Size([40000, 3, 32, 32])
Train labels shape:  torch.Size([40000])
Validation data shape:  torch.Size([10000, 3, 32, 32])
Validation labels shape:  torch.Size([10000])
Test data shape:  torch.Size([10000, 3, 32, 32])
Test labels shape:  torch.Size([10000])
```

3 Convolutional networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

4 Convolutional layer

As in the previous notebook, we will package each new neural network operator in a class that defines a `forward` and `backward` function.

4.1 Convolutional layer: forward

The core of a convolutional network is the convolution operation. Implement the forward pass for the convolution layer in the function `Conv.forward`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

After implementing the forward pass of the convolution operation, run the following to check your implementation. You should get a relative error less than `1e-7`.

```
[ ]: from convolutional_networks import Conv

x_shape = torch.tensor((2, 3, 4, 4))
w_shape = torch.tensor((3, 3, 4, 4))
x = torch.linspace(-0.1, 0.5, steps=torch.prod(x_shape), dtype=torch.float64,␣
 ↪device='cuda').reshape(*x_shape)
w = torch.linspace(-0.2, 0.3, steps=torch.prod(w_shape), dtype=torch.float64,␣
 ↪device='cuda').reshape(*w_shape)
b = torch.linspace(-0.1, 0.2, steps=3, dtype=torch.float64, device='cuda')

conv_param = {'stride': 2, 'pad': 1}
out, _ = Conv.forward(x, w, b, conv_param)
correct_out = torch.tensor([[[[-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]]]],
                          dtype=torch.float64, device='cuda',
            )

# Compare your output to ours; difference should be around e-8
print('Testing Conv.forward')
print('difference: ', ite4052.grad.rel_error(out, correct_out))
```

```
Testing Conv.forward
difference:  1.0141824738238694e-09
```

4.2 Aside: Image processing via convolutions

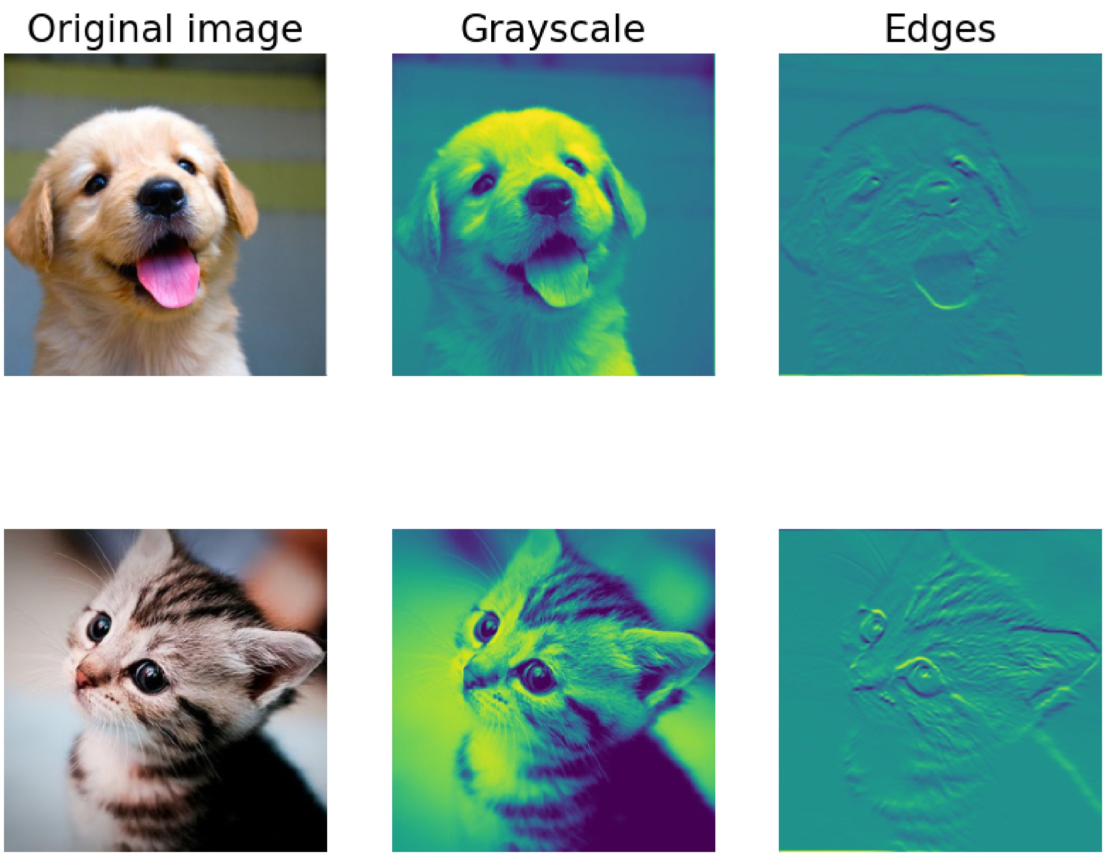
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

[ ]:
```
from imageio import imread
from PIL import Image
from torchvision.transforms import ToTensor

kitten_url = 'https://web.eecs.umich.edu/~justincj/teaching/eecs498/assets/a3/
 ↪kitten.jpg'
puppy_url = 'https://web.eecs.umich.edu/~justincj/teaching/eecs498/assets/a3/
 ↪puppy.jpg'

kitten = imread(kitten_url)
puppy = imread(puppy_url)
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_puppy = ToTensor()(Image.fromarray(puppy).resize((img_size, img_size)))
resized_kitten = ToTensor()(Image.fromarray(kitten_cropped).resize((img_size,␣
 ↪img_size)))
x = torch.stack([resized_puppy, resized_kitten])

# Set up a convolutional weights holding 2 filters, each 3x3
w = torch.zeros(2, 3, 3, 3, dtype=x.dtype)

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
```

```python
w[0, 0, :, :] = torch.tensor([[0, 0, 0], [0, 0.3, 0], [0, 0, 0]])
w[0, 1, :, :] = torch.tensor([[0, 0, 0], [0, 0.6, 0], [0, 0, 0]])
w[0, 2, :, :] = torch.tensor([[0, 0, 0], [0, 0.1, 0], [0, 0, 0]])

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = torch.tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = torch.tensor([0, 128], dtype=x.dtype)

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = Conv.forward(x, w, b, {'stride': 1, 'pad': 1})

def imshow_no_ax(img, normalize=True):
  """ Tiny helper to show images as uint8 and remove axis labels """
  if normalize:
    img_max, img_min = img.max(), img.min()
    img = 255.0 * (img - img_min) / (img_max - img_min)
  plt.imshow(img)
  plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_no_ax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_no_ax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_no_ax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_no_ax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_no_ax(out[1, 1])
plt.show()
```

```
<ipython-input-40-cced0ea9008c>:8: DeprecationWarning: Starting with ImageIO v3
the behavior of this function will switch to that of iio.v3.imread. To keep the
current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  kitten = imread(kitten_url)
```

```
<ipython-input-40-cced0ea9008c>:9: DeprecationWarning: Starting with ImageIO v3
the behavior of this function will switch to that of iio.v3.imread. To keep the
current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  puppy = imread(puppy_url)
```



4.3 Convolutional layer: backward

Implement the backward pass for the convolution operation in the function `Conv.backward`. Again, you don't need to worry too much about computational efficiency.

After implementing the convolution backward pass, run the following to test your implementation. You should get errors less than `1e-8`.

[ ]:
```python
from convolutional_networks import Conv

reset_seed(0)
x = torch.randn(4, 3, 5, 5, dtype=torch.float64, device='cuda')
w = torch.randn(2, 3, 3, 3, dtype=torch.float64, device='cuda')
b = torch.randn(2, dtype=torch.float64, device='cuda')
dout = torch.randn(4, 2, 5, 5, dtype=torch.float64, device='cuda')
```

```
conv_param = {'stride': 1, 'pad': 1}

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: Conv.forward(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = ite4052.grad.compute_numeric_gradient(lambda w: Conv.forward(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = ite4052.grad.compute_numeric_gradient(lambda b: Conv.forward(x, w, b,␣
 ↪conv_param)[0], b, dout)

out, cache = Conv.forward(x, w, b, conv_param)
dx, dw, db = Conv.backward(dout, cache)

print('Testing Conv.backward function')
print('dx error: ', ite4052.grad.rel_error(dx, dx_num))
print('dw error: ', ite4052.grad.rel_error(dw, dw_num))
print('db error: ', ite4052.grad.rel_error(db, db_num))
```

```
Testing Conv.backward function
dx error:  2.496440948929281e-09
dw error:  9.222783256472505e-10
db error:  1.201214303148521e-09
```

5 Max-pooling

5.1 Max-pooling: forward

Implement the forward pass for the max-pooling operation. Again, don't worry too much about computational efficiency.

After implementing the forward pass for max-pooling, run the following to check your implementation. You should get errors less than `1e-7`.

[78]:
```
from convolutional_networks import MaxPool

reset_seed(0)
x_shape = torch.tensor((2, 3, 4, 4))
x = torch.linspace(-0.3, 0.4, steps=torch.prod(x_shape), dtype=torch.float64,␣
 ↪device='cuda').reshape(*x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = MaxPool.forward(x, pool_param)

correct_out = torch.tensor([[[[-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       ]]]],
                           dtype=torch.float64, device='cuda')

# Compare your output with ours. Difference should be on the order of e-8.
print('Testing MaxPool.forward function:')
print('difference: ', ite4052.grad.rel_error(out, correct_out))
```

```
Testing MaxPool.forward function:
difference:  5.921052675939009e-09
```

5.2 Max-pooling: backward

Implement the backward pass for the max-pooling operation. You don't need to worry about computational efficiency.

Check your implementation of the max pooling backward pass with numeric gradient checking by running the following. You should get errors less than `1e-10`.

[79]:
```
from convolutional_networks import MaxPool

reset_seed(0)
x = torch.randn(3, 2, 8, 8, dtype=torch.float64, device='cuda')
dout = torch.randn(3, 2, 4, 4, dtype=torch.float64, device='cuda')
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = ite4052.grad.compute_numeric_gradient(lambda x: MaxPool.forward(x,␣
 ↪pool_param)[0], x, dout)

out, cache = MaxPool.forward(x, pool_param)
dx = MaxPool.backward(dout, cache)

print('Testing MaxPool.backward function:')
print('dx error: ', ite4052.grad.rel_error(dx, dx_num))
```

```
Testing MaxPool.backward function:
dx error:  6.653155794014975e-10
```

6 Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers. Those can be found at the bottom of `convolutional_networks.py`

The fast convolution implementation depends on `torch.nn`

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

```
class FastConv(object):

  @staticmethod
  def forward(x, w, b, conv_param):
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param['stride'], conv_param['pad']
    layer = torch.nn.Conv2d(C, F, (HH, WW), stride=stride, padding=pad)
    layer.weight = torch.nn.Parameter(w)
    layer.bias = torch.nn.Parameter(b)
    tx = x.detach()
    tx.requires_grad = True
    out = layer(tx)
    cache = (x, w, b, conv_param, tx, out, layer)
    return out, cache

  @staticmethod
  def backward(dout, cache):
    try:
      x, _, _, _, tx, out, layer = cache
      out.backward(dout)
      dx = tx.grad.detach()
      dw = layer.weight.grad.detach()
      db = layer.bias.grad.detach()
      layer.weight.grad = layer.bias.grad = None
    except RuntimeError:
      dx, dw, db = torch.zeros_like(tx), torch.zeros_like(layer.weight), torch.zeros_like(laye
    return dx, dw, db


class FastMaxPool(object):

  @staticmethod
  def forward(x, pool_param):
    N, C, H, W = x.shape
    pool_height, pool_width = pool_param['pool_height'], pool_param['pool_width']
    stride = pool_param['stride']
    layer = torch.nn.MaxPool2d(kernel_size=(pool_height, pool_width), stride=stride)
    tx = x.detach()
    tx.requires_grad = True
    out = layer(tx)
    cache = (x, pool_param, tx, out, layer)
    return out, cache
  @staticmethod
  def backward(dout, cache):
    try:
      x, _, tx, out, layer = cache
      out.backward(dout)
      dx = tx.grad.detach()
    except RuntimeError:
      dx = torch.zeros_like(tx)
    return dx
```

We will now compare three different implementations of convolution (both forward and backward):

1. Your naive, non-vectorized implementation on CPU
2. The fast, vectorized implementation on CPU
3. The fast, vectorized implementation on GPU

The differences between your implementation and FastConv should be less than `1e-10`. When moving from your implementation to FastConv CPU, you will likely see speedups of at least 100x. When comparing your implementation to FastConv CUDA, you will likely see speedups of more than 500x. (These speedups are not hard requirements for this assignment since we are not asking you to write any vectorized implementations)

[ ]:
```
# Rel errors should be around e-11 or less
from convolutional_networks import Conv, FastConv

reset_seed(0)
x = torch.randn(10, 3, 31, 31, dtype=torch.float64, device='cuda')
w = torch.randn(25, 3, 3, 3, dtype=torch.float64, device='cuda')
b = torch.randn(25, dtype=torch.float64, device='cuda')
dout = torch.randn(10, 25, 16, 16, dtype=torch.float64, device='cuda')
x_cuda, w_cuda, b_cuda, dout_cuda = x.to('cuda'), w.to('cuda'), b.to('cuda'),␣
 ↪dout.to('cuda')
conv_param = {'stride': 2, 'pad': 1}

t0 = time.time()
out_naive, cache_naive = Conv.forward(x, w, b, conv_param)
t1 = time.time()
out_fast, cache_fast = FastConv.forward(x, w, b, conv_param)
t2 = time.time()
out_fast_cuda, cache_fast_cuda = FastConv.forward(x_cuda, w_cuda, b_cuda,␣
 ↪conv_param)
t3 = time.time()

print('Testing FastConv.forward:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Fast CUDA: %fs' % (t3 - t2))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Speedup CUDA: %fx' % ((t1 - t0) / (t3 - t2)))
print('Difference: ', ite4052.grad.rel_error(out_naive, out_fast))
print('Difference CUDA: ', ite4052.grad.rel_error(out_naive, out_fast_cuda.
 ↪to(out_naive.device)))

t0 = time.time()
dx_naive, dw_naive, db_naive = Conv.backward(dout, cache_naive)
t1 = time.time()
dx_fast, dw_fast, db_fast = FastConv.backward(dout, cache_fast)
t2 = time.time()
dx_fast_cuda, dw_fast_cuda, db_fast_cuda = FastConv.backward(dout_cuda,␣
 ↪cache_fast_cuda)
t3 = time.time()

print('\nTesting FastConv.backward:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Fast CUDA: %fs' % (t3 - t2))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Speedup CUDA: %fx' % ((t1 - t0) / (t3 - t2)))
print('dx difference: ', ite4052.grad.rel_error(dx_naive, dx_fast))
print('dw difference: ', ite4052.grad.rel_error(dw_naive, dw_fast))
print('db difference: ', ite4052.grad.rel_error(db_naive, db_fast))
print('dx difference CUDA: ', ite4052.grad.rel_error(dx_naive, dx_fast_cuda.
 ↪to(dx_naive.device)))
print('dw difference CUDA: ', ite4052.grad.rel_error(dw_naive, dw_fast_cuda.
 ↪to(dw_naive.device)))
print('db difference CUDA: ', ite4052.grad.rel_error(db_naive, db_fast_cuda.
 ↪to(db_naive.device)))
```

```
Testing FastConv.forward:
Naive: 5.833348s
Fast: 0.001049s
Fast CUDA: 0.000471s
Speedup: 5561.907934x
Speedup CUDA: 12394.545593x
Difference:  2.1928544370986248e-16
Difference CUDA:  2.1928544370986248e-16

Testing FastConv.backward:
Naive: 6.889890s
Fast: 0.007658s
Fast CUDA: 0.000604s
Speedup: 899.697821x
Speedup CUDA: 11399.721499x
dx difference:  3.8774834488572664e-16
dw difference:  1.0239360372210248e-15
```

```
db difference:  1.6798889889341262e-16
dx difference CUDA:  3.8774834488572664e-16
dw difference CUDA:  1.0239360372210248e-15
db difference CUDA:  1.6798889889341262e-16
```

We will now similarly compare your naive implementation of max pooling against the fast implementation. You should see differences of 0 between your implementation and the fast implementation.

When comparing your implementation against FastMaxPool on CPU, you will likely see speedups of more than 100x. When comparing your implementation against FastMaxPool on GPU, you will likely see speedups of more than 500x.

[ ]:
```
# Relative errors should be close to 0.0
from convolutional_networks import Conv, MaxPool, FastConv, FastMaxPool



reset_seed(0)
x = torch.randn(40, 3, 32, 32, dtype=torch.float64, device='cuda')
dout = torch.randn(40, 3, 16, 16, dtype=torch.float64, device='cuda')
x_cuda, dout_cuda = x.to('cuda'), dout.to('cuda')
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time.time()
out_naive, cache_naive = MaxPool.forward(x, pool_param)
t1 = time.time()
out_fast, cache_fast = FastMaxPool.forward(x, pool_param)
t2 = time.time()
out_fast_cuda, cache_fast_cuda = FastMaxPool.forward(x_cuda, pool_param)
t3 = time.time()

print('Testing FastMaxPool.forward:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Fast CUDA: %fs' % (t3 - t2))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Speedup CUDA: %fx' % ((t1 - t0) / (t3 - t2)))
print('Difference: ', ite4052.grad.rel_error(out_naive, out_fast))
print('Difference CUDA: ', ite4052.grad.rel_error(out_naive, out_fast_cuda.
  ↪to(out_naive.device)))

t0 = time.time()
dx_naive = MaxPool.backward(dout, cache_naive)
t1 = time.time()
dx_fast = FastMaxPool.backward(dout, cache_fast)
t2 = time.time()
dx_fast_cuda = FastMaxPool.backward(dout_cuda, cache_fast_cuda)
t3 = time.time()
```

```
print('\nTesting FastMaxPool.backward:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Fast CUDA: %fs' % (t3 - t2))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Speedup CUDA: %fx' % ((t1 - t0) / (t3 - t2)))
print('dx difference: ', ite4052.grad.rel_error(dx_naive, dx_fast))
print('dx difference CUDA: ', ite4052.grad.rel_error(dx_naive, dx_fast_cuda.
  ↪to(dx_naive.device)))
```

```
Testing FastMaxPool.forward:
Naive: 1.487928s
Fast: 0.000321s
Fast CUDA: 0.000179s
Speedup: 4636.569837x
Speedup CUDA: 8298.966755x
Difference:  0.0
Difference CUDA:  0.0

Testing FastMaxPool.backward:
Naive: 5.024980s
Fast: 0.000483s
Fast CUDA: 0.000330s
Speedup: 10397.776517x
Speedup CUDA: 15217.540072x
dx difference:  0.0
dx difference CUDA:  0.0
```

7 Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. Below you will find sandwich layers that implement a few commonly used patterns for convolutional networks. We've included them at the bottom of `covolutional_networks.py` Run the cells below to sanity check they're working.

Note: This will be using the ReLU function you implemented in the previous notebook. Make sure to implement it first.

```
class Conv_ReLU(object):

  @staticmethod
  def forward(x, w, b, conv_param):
    """
    A convenience layer that performs a convolution followed by a ReLU.
    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer
    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    a, conv_cache = FastConv.forward(x, w, b, conv_param)
    out, relu_cache = ReLU.forward(a)
    cache = (conv_cache, relu_cache)
    return out, cache

  @staticmethod
  def backward(dout, cache):
    """
    Backward pass for the conv-relu convenience layer.
    """
    conv_cache, relu_cache = cache
    da = ReLU.backward(dout, relu_cache)
    dx, dw, db = FastConv.backward(da, conv_cache)
    return dx, dw, db


class Conv_ReLU_Pool(object):

  @staticmethod
  def forward(x, w, b, conv_param, pool_param):
    """
    A convenience layer that performs a convolution, a ReLU, and a pool.
    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer
    - pool_param: Parameters for the pooling layer
    Returns a tuple of:
    - out: Output from the pooling layer
    - cache: Object to give to the backward pass
    """
    a, conv_cache = FastConv.forward(x, w, b, conv_param)
    s, relu_cache = ReLU.forward(a)
    out, pool_cache = FastMaxPool.forward(s, pool_param)
    cache = (conv_cache, relu_cache, pool_cache)
    return out, cache

  @staticmethod
  def backward(dout, cache):
    """
    Backward pass for the conv-relu-pool convenience layer
    """
    conv_cache, relu_cache, pool_cache = cache
    ds = FastMaxPool.backward(dout, pool_cache)
    da = ReLU.backward(ds, relu_cache)
    dx, dw, db = FastConv.backward(da, conv_cache)
    return dx, dw, db
```

Test the implementations of the sandwich layers by running the following. You should see errors less than `1e-7`.

```
[ ]: from convolutional_networks import Conv_ReLU, Conv_ReLU_Pool
     reset_seed(0)

     # Test Conv ReLU
     x = torch.randn(2, 3, 8, 8, dtype=torch.float64, device='cuda')
     w = torch.randn(3, 3, 3, 3, dtype=torch.float64, device='cuda')
     b = torch.randn(3, dtype=torch.float64, device='cuda')
     dout = torch.randn(2, 3, 8, 8, dtype=torch.float64, device='cuda')
     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 = ite4052.grad.compute_numeric_gradient(lambda x: Conv_ReLU.forward(x,␣
      ↪w, b, conv_param)[0], x, dout)
     dw_num = ite4052.grad.compute_numeric_gradient(lambda w: Conv_ReLU.forward(x,␣
      ↪w, b, conv_param)[0], w, dout)
     db_num = ite4052.grad.compute_numeric_gradient(lambda b: Conv_ReLU.forward(x,␣
      ↪w, b, conv_param)[0], b, dout)

     # Relative errors should be around e-8 or less
     print('Testing Conv_ReLU:')
     print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
     print('dw error: ', ite4052.grad.rel_error(dw_num, dw))
     print('db error: ', ite4052.grad.rel_error(db_num, db))

     # Test Conv ReLU Pool
     x = torch.randn(2, 3, 16, 16, dtype=torch.float64, device='cuda')
     w = torch.randn(3, 3, 3, 3, dtype=torch.float64, device='cuda')
     b = torch.randn(3, dtype=torch.float64, device='cuda')
     dout = torch.randn(2, 3, 8, 8, dtype=torch.float64, device='cuda')
     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 = ite4052.grad.compute_numeric_gradient(lambda x: Conv_ReLU_Pool.
      ↪forward(x, w, b, conv_param, pool_param)[0], x, dout)
     dw_num = ite4052.grad.compute_numeric_gradient(lambda w: Conv_ReLU_Pool.
      ↪forward(x, w, b, conv_param, pool_param)[0], w, dout)
     db_num = ite4052.grad.compute_numeric_gradient(lambda b: Conv_ReLU_Pool.
      ↪forward(x, w, b, conv_param, pool_param)[0], b, dout)
```

```
# Relative errors should be around e-8 or less
print()
print('Testing Conv_ReLU_Pool')
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
print('dw error: ', ite4052.grad.rel_error(dw_num, dw))
print('db error: ', ite4052.grad.rel_error(db_num, db))
```

```
Testing Conv_ReLU:
dx error:  1.8037001509296748e-09
dw error:  1.2470995998634857e-09
db error:  1.1230402096612364e-09

Testing Conv_ReLU_Pool
dx error:  1.5915037060449427e-09
dw error:  1.8962680214651407e-09
db error:  5.05984212319748e-09
```

8 Three-layer convolutional network

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Complete the implementation of the `ThreeLayerConvNet` class. We STRONGLY recommend you to use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

8.1 Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization the loss should go up slightly.

[ ]:
```
from convolutional_networks import ThreeLayerConvNet

reset_seed(0)
model = ThreeLayerConvNet(dtype=torch.float64, device='cuda')

N = 50
X = torch.randn(N, 3, 32, 32, dtype=torch.float64, device='cuda')
y = torch.randint(10, size=(N,), dtype=torch.int64, device='cuda')

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss.item())

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss.item())
```

```
Initial loss (no regularization):  2.302584070548802
Initial loss (with regularization):  2.715361160771427
```

8.2 Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artificial data and a small number of neurons at each layer.

You should see errors less than 1e-5.

```
[ ]: from convolutional_networks import ThreeLayerConvNet

     num_inputs = 2
     input_dims = (3, 16, 16)
     reg = 0.0
     num_classes = 10
     reset_seed(0)
     X = torch.randn(num_inputs, *input_dims, dtype=torch.float64, device='cuda')
     y = torch.randint(num_classes, size=(num_inputs,), dtype=torch.int64,␣
      ↪device='cuda')

     model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                               input_dims=input_dims, hidden_dim=7,
                               weight_scale=5e-2, dtype=torch.float64, device='cuda')
     loss, grads = model.loss(X, y)

     for param_name in sorted(grads):
         f = lambda _: model.loss(X, y)[0]
         param_grad_num = ite4052.grad.compute_numeric_gradient(f, model.
      ↪params[param_name])
         print('%s max relative error: %e' % (param_name, ite4052.grad.
      ↪rel_error(param_grad_num, grads[param_name])))
```

```
W1 max relative error: 2.851376e-08
W2 max relative error: 7.622434e-08
W3 max relative error: 4.168769e-09
b1 max relative error: 1.252231e-08
b2 max relative error: 1.551252e-08
b3 max relative error: 3.205744e-09
```

8.3 Overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
[ ]: from convolutional_networks import ThreeLayerConvNet
     from fully_connected_networks import adam
```

```
reset_seed(0)

num_train = 100
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-3, dtype=torch.float32, device='cuda')

solver = Solver(model, small_data,
                num_epochs=30, batch_size=50,
                update_rule=adam,
                optim_config={
                  'learning_rate': 2e-3,
                },
                verbose=True, print_every=1,
                device='cuda')
solver.train()
```

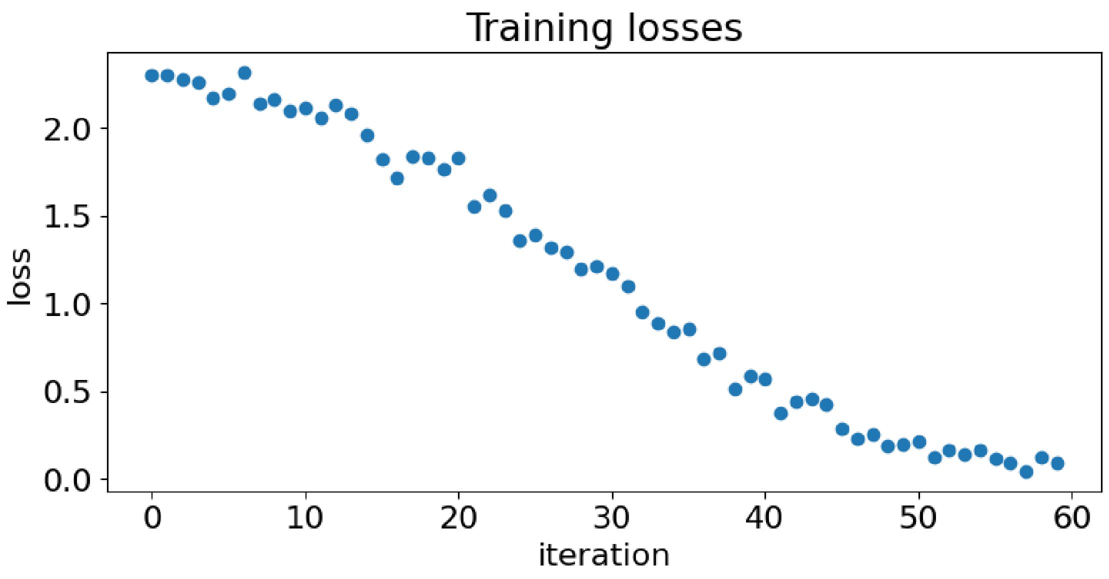
```
(Time 0.01 sec; Iteration 1 / 60) loss: 2.302585
(Epoch 0 / 30) train acc: 0.160000; val_acc: 0.101400
(Time 0.25 sec; Iteration 2 / 60) loss: 2.299526
(Epoch 1 / 30) train acc: 0.160000; val_acc: 0.101400
(Time 0.44 sec; Iteration 3 / 60) loss: 2.280210
(Time 0.45 sec; Iteration 4 / 60) loss: 2.265939
(Epoch 2 / 30) train acc: 0.160000; val_acc: 0.101400
(Time 0.64 sec; Iteration 5 / 60) loss: 2.171782
(Time 0.64 sec; Iteration 6 / 60) loss: 2.199671
(Epoch 3 / 30) train acc: 0.160000; val_acc: 0.101400
(Time 0.83 sec; Iteration 7 / 60) loss: 2.321307
(Time 0.84 sec; Iteration 8 / 60) loss: 2.136467
(Epoch 4 / 30) train acc: 0.310000; val_acc: 0.136700
(Time 1.04 sec; Iteration 9 / 60) loss: 2.165079
(Time 1.04 sec; Iteration 10 / 60) loss: 2.102443
(Epoch 5 / 30) train acc: 0.320000; val_acc: 0.147000
(Time 1.24 sec; Iteration 11 / 60) loss: 2.113706
(Time 1.25 sec; Iteration 12 / 60) loss: 2.060869
(Epoch 6 / 30) train acc: 0.340000; val_acc: 0.151900
(Time 1.44 sec; Iteration 13 / 60) loss: 2.128923
(Time 1.45 sec; Iteration 14 / 60) loss: 2.079929
(Epoch 7 / 30) train acc: 0.350000; val_acc: 0.158100
(Time 1.65 sec; Iteration 15 / 60) loss: 1.964534
(Time 1.65 sec; Iteration 16 / 60) loss: 1.822080
```

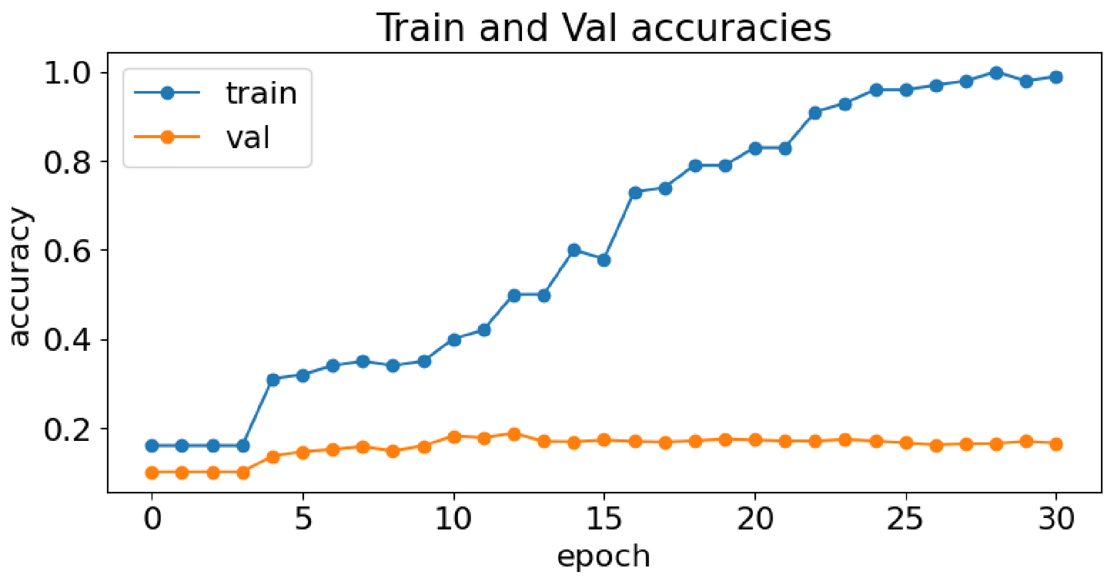
```
(Epoch 8 / 30) train acc: 0.340000; val_acc: 0.148500
(Time 1.85 sec; Iteration 17 / 60) loss: 1.717725
(Time 1.86 sec; Iteration 18 / 60) loss: 1.836852
(Epoch 9 / 30) train acc: 0.350000; val_acc: 0.159800
(Time 2.05 sec; Iteration 19 / 60) loss: 1.827867
(Time 2.06 sec; Iteration 20 / 60) loss: 1.763564
(Epoch 10 / 30) train acc: 0.400000; val_acc: 0.181900
(Time 2.26 sec; Iteration 21 / 60) loss: 1.832894
(Time 2.26 sec; Iteration 22 / 60) loss: 1.553279
(Epoch 11 / 30) train acc: 0.420000; val_acc: 0.178000
(Time 2.46 sec; Iteration 23 / 60) loss: 1.623682
(Time 2.47 sec; Iteration 24 / 60) loss: 1.526920
(Epoch 12 / 30) train acc: 0.500000; val_acc: 0.187900
(Time 2.66 sec; Iteration 25 / 60) loss: 1.360784
(Time 2.67 sec; Iteration 26 / 60) loss: 1.394990
(Epoch 13 / 30) train acc: 0.500000; val_acc: 0.169900
(Time 2.87 sec; Iteration 27 / 60) loss: 1.321774
(Time 2.87 sec; Iteration 28 / 60) loss: 1.294489
(Epoch 14 / 30) train acc: 0.600000; val_acc: 0.169100
(Time 3.07 sec; Iteration 29 / 60) loss: 1.198380
(Time 3.08 sec; Iteration 30 / 60) loss: 1.214284
(Epoch 15 / 30) train acc: 0.580000; val_acc: 0.172900
(Time 3.28 sec; Iteration 31 / 60) loss: 1.170424
(Time 3.29 sec; Iteration 32 / 60) loss: 1.101528
(Epoch 16 / 30) train acc: 0.730000; val_acc: 0.170000
(Time 3.48 sec; Iteration 33 / 60) loss: 0.950446
(Time 3.49 sec; Iteration 34 / 60) loss: 0.887184
(Epoch 17 / 30) train acc: 0.740000; val_acc: 0.168300
(Time 3.68 sec; Iteration 35 / 60) loss: 0.841856
(Time 3.69 sec; Iteration 36 / 60) loss: 0.853522
(Epoch 18 / 30) train acc: 0.790000; val_acc: 0.171300
(Time 3.88 sec; Iteration 37 / 60) loss: 0.683822
(Time 3.89 sec; Iteration 38 / 60) loss: 0.717242
(Epoch 19 / 30) train acc: 0.790000; val_acc: 0.175300
(Time 4.09 sec; Iteration 39 / 60) loss: 0.514168
(Time 4.09 sec; Iteration 40 / 60) loss: 0.587726
(Epoch 20 / 30) train acc: 0.830000; val_acc: 0.173100
(Time 4.29 sec; Iteration 41 / 60) loss: 0.573341
(Time 4.29 sec; Iteration 42 / 60) loss: 0.373067
(Epoch 21 / 30) train acc: 0.830000; val_acc: 0.170900
(Time 4.49 sec; Iteration 43 / 60) loss: 0.444458
(Time 4.49 sec; Iteration 44 / 60) loss: 0.457126
(Epoch 22 / 30) train acc: 0.910000; val_acc: 0.170700
(Time 4.69 sec; Iteration 45 / 60) loss: 0.420783
(Time 4.69 sec; Iteration 46 / 60) loss: 0.288334
(Epoch 23 / 30) train acc: 0.930000; val_acc: 0.174300
(Time 4.89 sec; Iteration 47 / 60) loss: 0.226658
(Time 4.89 sec; Iteration 48 / 60) loss: 0.255359
```

```
(Epoch 24 / 30) train acc: 0.960000; val_acc: 0.170900
(Time 5.09 sec; Iteration 49 / 60) loss: 0.189274
(Time 5.10 sec; Iteration 50 / 60) loss: 0.195910
(Epoch 25 / 30) train acc: 0.960000; val_acc: 0.166300
(Time 5.29 sec; Iteration 51 / 60) loss: 0.211350
(Time 5.30 sec; Iteration 52 / 60) loss: 0.125847
(Epoch 26 / 30) train acc: 0.970000; val_acc: 0.162000
(Time 5.49 sec; Iteration 53 / 60) loss: 0.165948
(Time 5.50 sec; Iteration 54 / 60) loss: 0.142868
(Epoch 27 / 30) train acc: 0.980000; val_acc: 0.164400
(Time 5.69 sec; Iteration 55 / 60) loss: 0.161026
(Time 5.70 sec; Iteration 56 / 60) loss: 0.117419
(Epoch 28 / 30) train acc: 1.000000; val_acc: 0.164900
(Time 5.89 sec; Iteration 57 / 60) loss: 0.094198
(Time 5.90 sec; Iteration 58 / 60) loss: 0.044470
(Epoch 29 / 30) train acc: 0.980000; val_acc: 0.170000
(Time 6.09 sec; Iteration 59 / 60) loss: 0.124258
(Time 6.10 sec; Iteration 60 / 60) loss: 0.093202
(Epoch 30 / 30) train acc: 0.990000; val_acc: 0.165900
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

[ ]:
```
plt.title('Training losses')
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')
plt.gcf().set_size_inches(9, 4)
plt.show()

plt.title('Train and Val accuracies')
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.gcf().set_size_inches(9, 4)
plt.show()
```





8.4 Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 50% accuracy on the training set:

[ ]:
```
from convolutional_networks import ThreeLayerConvNet
from fully_connected_networks import adam

reset_seed(0)
```

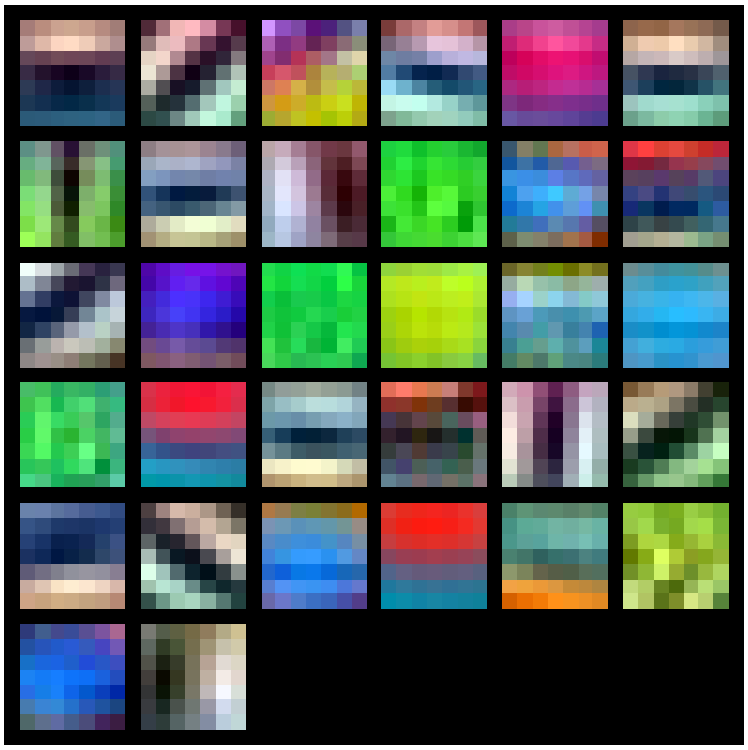
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001,␣
 ↪dtype=torch.float, device='cuda')

solver = Solver(model, data_dict,
                num_epochs=1, batch_size=64,
                update_rule=adam,
                optim_config={
                  'learning_rate': 2e-3,
                },
                verbose=True, print_every=50, device='cuda')
solver.train()
```

```
(Time 0.01 sec; Iteration 1 / 625) loss: 2.306690
(Epoch 0 / 1) train acc: 0.105000; val_acc: 0.102500
(Time 0.62 sec; Iteration 51 / 625) loss: 2.244238
(Time 1.02 sec; Iteration 101 / 625) loss: 2.152954
(Time 1.40 sec; Iteration 151 / 625) loss: 1.838577
(Time 1.79 sec; Iteration 201 / 625) loss: 1.761443
(Time 2.17 sec; Iteration 251 / 625) loss: 1.723239
(Time 2.56 sec; Iteration 301 / 625) loss: 1.730495
(Time 2.95 sec; Iteration 351 / 625) loss: 1.599747
(Time 3.33 sec; Iteration 401 / 625) loss: 1.642913
(Time 3.72 sec; Iteration 451 / 625) loss: 1.666422
(Time 4.10 sec; Iteration 501 / 625) loss: 1.590931
(Time 4.49 sec; Iteration 551 / 625) loss: 1.608762
(Time 4.88 sec; Iteration 601 / 625) loss: 1.568186
(Epoch 1 / 1) train acc: 0.541000; val_acc: 0.516200
```

8.5 Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

```
[ ]: from torchvision.utils import make_grid
     nrow = math.ceil(math.sqrt(model.params['W1'].shape[0]))
     grid = make_grid(model.params['W1'], nrow=nrow, padding=1, normalize=True,␣
      ↪scale_each=True)
     plt.imshow(grid.to(device='cpu').permute(1, 2, 0))
     plt.axis('off')
     plt.gcf().set_size_inches(5, 5)
     plt.show()
```



9 Deep convolutional network

Next you will implement a deep convolutional network with an arbitrary number of conv layers in VGGNet style.

Read through the `DeepConvNet` class.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing batch normalization; we will add those features soon. Again, we STRONGLY recommend you to use the fast/sandwich layers (already imported for you) in your implementation.

9.1 Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization the loss should go up slightly.

[ ]:
```
from convolutional_networks import DeepConvNet
from fully_connected_networks import adam

reset_seed(0)
input_dims = (3, 32, 32)
```

```
model = DeepConvNet(num_filters=[8, 64], max_pools=[0, 1], dtype=torch.float64,␣
  ↪device='cuda')

N = 50
X = torch.randn(N, *input_dims, dtype=torch.float64, device='cuda')
y = torch.randint(10, size=(N,), dtype=torch.int64, device='cuda')

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss.item())

model.reg = 1.
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss.item())
```

```
Initial loss (no regularization):  2.302584572517059
Initial loss (with regularization):  2.3482176522105846
```

9.2 Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer.

You should see relative errors less than 1e-5.

```
[ ]: from convolutional_networks import DeepConvNet
     from fully_connected_networks import adam

     reset_seed(0)
     num_inputs = 2
     input_dims = (3, 8, 8)
     num_classes = 10
     X = torch.randn(N, *input_dims, dtype=torch.float64, device='cuda')
     y = torch.randint(10, size=(N,), dtype=torch.int64, device='cuda')

     for reg in [0, 3.14]:
       print('Running check with reg = ', reg)
       model = DeepConvNet(input_dims=input_dims, num_classes=num_classes,
                           num_filters=[8, 8, 8],
                           max_pools=[0, 2],
                           reg=reg,
                           weight_scale=5e-2, dtype=torch.float64, device='cuda')

       loss, grads = model.loss(X, y)
       # The relative errors should be up to the order of e-6
       for name in sorted(grads):
         f = lambda _: model.loss(X, y)[0]
         grad_num = ite4052.grad.compute_numeric_gradient(f, model.params[name])
```

```
    print('%s max relative error: %e' % (name, ite4052.grad.rel_error(grad_num,␣
↪grads[name])))
  if reg == 0: print()
```

```
Running check with reg =  0
W1 max relative error: 5.912415e-07
W2 max relative error: 8.190183e-07
W3 max relative error: 5.317616e-07
W4 max relative error: 6.344352e-07
b1 max relative error: 6.956690e-07
b2 max relative error: 4.282076e-07
b3 max relative error: 2.831506e-08
b4 max relative error: 1.407349e-08

Running check with reg =  3.14
W1 max relative error: 7.242144e-09
W2 max relative error: 1.032423e-08
W3 max relative error: 1.289720e-08
W4 max relative error: 7.251152e-09
b1 max relative error: 1.994212e-06
b2 max relative error: 1.289978e-06
b3 max relative error: 4.524272e-07
b4 max relative error: 9.903249e-08
```

9.3 Overfit small data

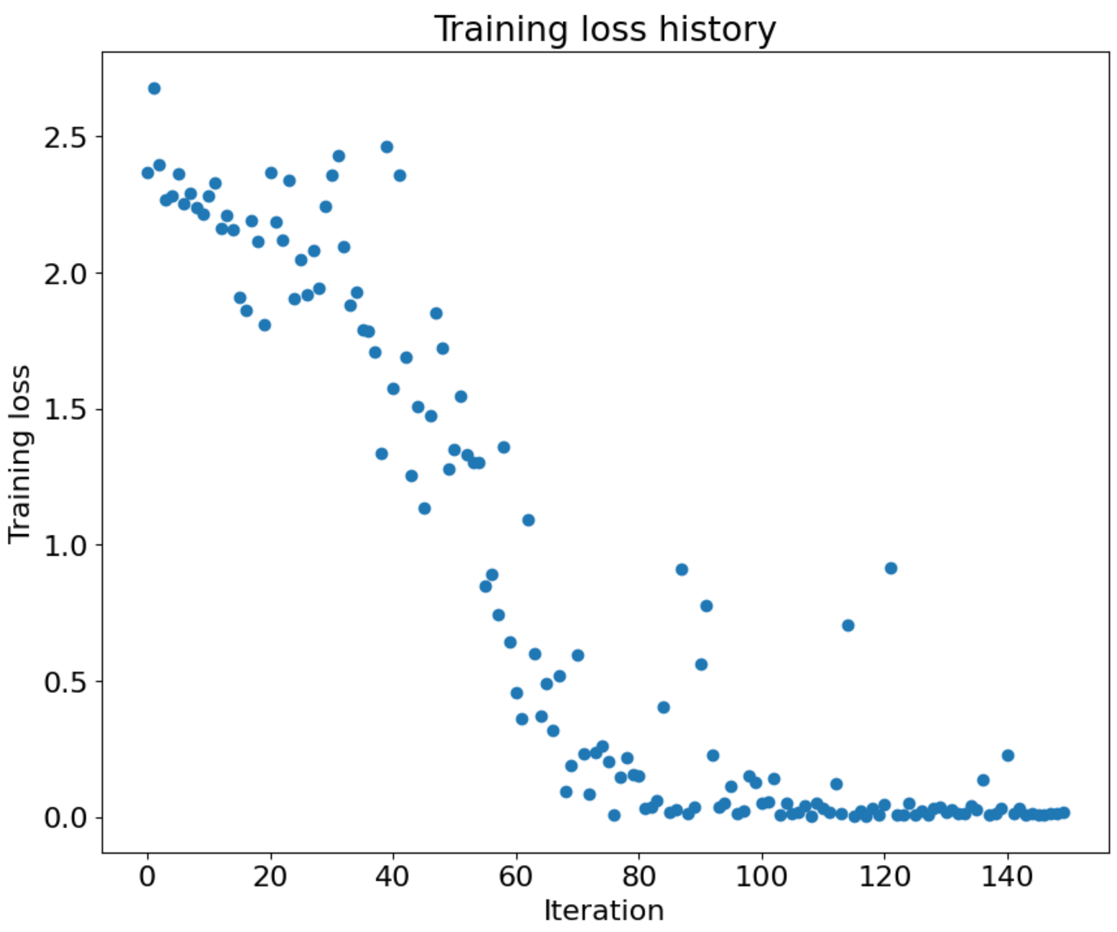
As another sanity check, make sure you can overfit a small dataset of 50 images. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy within 30 epochs.

[ ]:
```
# TODO: Use a DeepConvNet to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.
from convolutional_networks import DeepConvNet, find_overfit_parameters
from fully_connected_networks import adam

reset_seed(0)
num_train = 50
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}
input_dims = small_data['X_train'].shape[1:]



# Update the parameters in find_overfit_parameters in convolutional_networks.py
weight_scale, learning_rate = find_overfit_parameters()
```

```
model = DeepConvNet(input_dims=input_dims, num_classes=10,
                    num_filters=[8, 16, 32, 64],
                    max_pools=[0, 1, 2, 3],
                    reg=1e-5, weight_scale=weight_scale, dtype=torch.float32,␣
 ↪device='cuda')
solver = Solver(model, small_data,
                print_every=10, num_epochs=30, batch_size=10,
                update_rule=adam,
                optim_config={
                  'learning_rate': learning_rate,
                },
                device='cuda',
         )
# Turn off keep_best_params to allow final weights to be saved, instead of best␣
 ↪weights on validation set.
solver.train(return_best_params=False)

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()

val_acc = solver.check_accuracy(
                        solver.X_train, solver.y_train, num_samples=solver.
 ↪num_train_samples
                    )
print(val_acc)
```

```
(Time 0.02 sec; Iteration 1 / 150) loss: 2.366917
(Epoch 0 / 30) train acc: 0.160000; val_acc: 0.101600
(Epoch 1 / 30) train acc: 0.120000; val_acc: 0.101400
(Epoch 2 / 30) train acc: 0.120000; val_acc: 0.101400
(Time 0.86 sec; Iteration 11 / 150) loss: 2.281695
(Epoch 3 / 30) train acc: 0.200000; val_acc: 0.118900
(Epoch 4 / 30) train acc: 0.160000; val_acc: 0.095200
(Time 1.47 sec; Iteration 21 / 150) loss: 2.367848
(Epoch 5 / 30) train acc: 0.280000; val_acc: 0.156200
(Epoch 6 / 30) train acc: 0.320000; val_acc: 0.134800
(Time 2.06 sec; Iteration 31 / 150) loss: 2.360043
(Epoch 7 / 30) train acc: 0.480000; val_acc: 0.124600
(Epoch 8 / 30) train acc: 0.400000; val_acc: 0.163600
(Time 3.22 sec; Iteration 41 / 150) loss: 1.577206
(Epoch 9 / 30) train acc: 0.600000; val_acc: 0.158100
(Epoch 10 / 30) train acc: 0.680000; val_acc: 0.156100
```

```
(Time 3.81 sec; Iteration 51 / 150) loss: 1.351161
(Epoch 11 / 30) train acc: 0.800000; val_acc: 0.176100
(Epoch 12 / 30) train acc: 0.880000; val_acc: 0.153900
(Time 4.23 sec; Iteration 61 / 150) loss: 0.457363
(Epoch 13 / 30) train acc: 0.960000; val_acc: 0.156300
(Epoch 14 / 30) train acc: 0.900000; val_acc: 0.167600
(Time 4.65 sec; Iteration 71 / 150) loss: 0.596891
(Epoch 15 / 30) train acc: 0.960000; val_acc: 0.149500
(Epoch 16 / 30) train acc: 0.940000; val_acc: 0.163000
(Time 5.06 sec; Iteration 81 / 150) loss: 0.153083
(Epoch 17 / 30) train acc: 0.940000; val_acc: 0.149200
(Epoch 18 / 30) train acc: 0.900000; val_acc: 0.164500
(Time 5.49 sec; Iteration 91 / 150) loss: 0.562661
(Epoch 19 / 30) train acc: 0.960000; val_acc: 0.145500
(Epoch 20 / 30) train acc: 0.980000; val_acc: 0.146200
(Time 5.92 sec; Iteration 101 / 150) loss: 0.050986
(Epoch 21 / 30) train acc: 1.000000; val_acc: 0.168000
(Epoch 22 / 30) train acc: 0.960000; val_acc: 0.172900
(Time 6.33 sec; Iteration 111 / 150) loss: 0.033450
(Epoch 23 / 30) train acc: 0.980000; val_acc: 0.175700
(Epoch 24 / 30) train acc: 0.980000; val_acc: 0.166000
(Time 6.76 sec; Iteration 121 / 150) loss: 0.047047
(Epoch 25 / 30) train acc: 1.000000; val_acc: 0.172000
(Epoch 26 / 30) train acc: 1.000000; val_acc: 0.172600
(Time 7.18 sec; Iteration 131 / 150) loss: 0.017503
(Epoch 27 / 30) train acc: 0.980000; val_acc: 0.147700
(Epoch 28 / 30) train acc: 0.960000; val_acc: 0.157500
(Time 7.61 sec; Iteration 141 / 150) loss: 0.225743
(Epoch 29 / 30) train acc: 1.000000; val_acc: 0.158500
(Epoch 30 / 30) train acc: 1.000000; val_acc: 0.146700
```



```
1.0
```

If you're happy with the model's perfromance, run the following cell to save it.

We will also reload the model and run it on the training data to verify it's the right weights.

```
[ ]: path = os.path.join(GOOGLE_DRIVE_PATH, 'overfit_deepconvnet.pth')
     solver.model.save(path)

     # Create a new instance
     model = DeepConvNet(input_dims=input_dims, num_classes=10,
                         num_filters=[8, 16, 32, 64],
                         max_pools=[0, 1, 2, 3],
                         reg=1e-5, weight_scale=weight_scale, dtype=torch.float32,␣
      ↪device='cuda')
     solver = Solver(model, small_data,
                     print_every=10, num_epochs=30, batch_size=10,
                     update_rule=adam,
                     optim_config={
```

```
                  'learning_rate': learning_rate,
                },
                device='cuda',
        )


# Load model
solver.model.load(path, dtype=torch.float32, device='cuda')

# Evaluate on validation set
accuracy = solver.check_accuracy(small_data['X_train'], small_data['y_train'])
print(f"Saved model's accuracy on training is {accuracy}")
```

```
Saved in drive/My Drive/ITE4052/A4/overfit_deepconvnet.pth
load checkpoint file: drive/My Drive/ITE4052/A4/overfit_deepconvnet.pth
Saved model's accuracy on training is 1.0
```

10 Kaiming initialization

So far, you manually tuned the weight scale and for weight initialization. However, this is inefficient when it comes to training deep neural networks; practically, as your weight matrix is larger, the weight scale should be small. Below you will implement Kaiming initialization. For more details, refer to cs231n note and PyTorch documentation.

11 Convolutional nets with Kaiming initialization

Now that you have a working implementation for Kaiming initialization, go back to your Deep-Convnet. Modify your implementation to add Kaiming initialization.

Concretely, when the `weight_scale` is set to `'kaiming'` in the constructor, you should initialize weights of convolutional and linear layers using `kaiming_initializer`. Once you are done, run the following to see the effect of kaiming initialization in deep CNNs.

In this experiment, we train a 31-layer network with four different weight initialization schemes. Among them, only the Kaiming initialization method should achieve a non-random accuracy after one epoch of training.

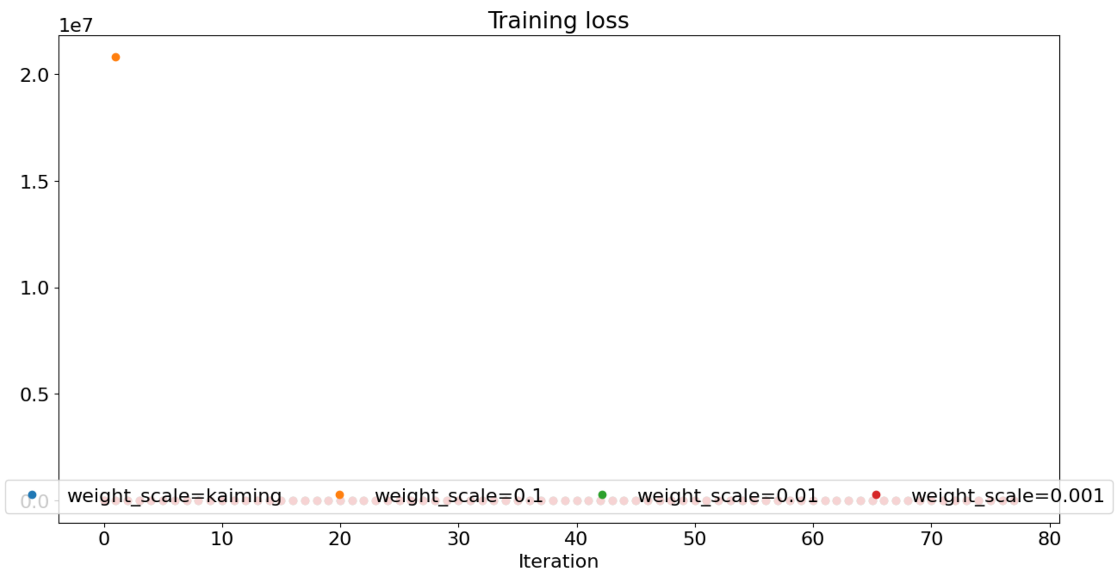
You may see `nan` loss when `weight_scale` is large, this shows a catastrophe of inappropriate weight initialization.

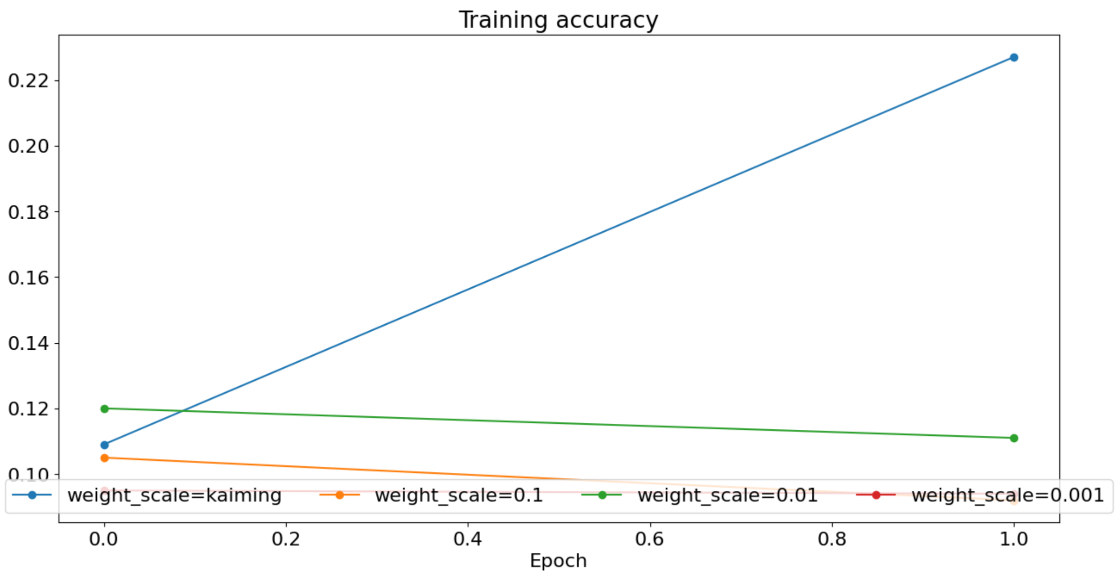
[ ]:
```
from convolutional_networks import DeepConvNet
from fully_connected_networks import sgd_momentum
reset_seed(0)

# Try training a deep convolutional net with different weight initialization␣
  ↪methods
num_train = 10000
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}
input_dims = data_dict['X_train'].shape[1:]

weight_scales = ['kaiming', 1e-1, 1e-2, 1e-3]

solvers = []
for weight_scale in weight_scales:
  print('Solver with weight scale: ', weight_scale)
  model = DeepConvNet(input_dims=input_dims, num_classes=10,
                    num_filters=([8] * 10) + ([32] * 10) + ([128] * 10),
                    max_pools=[9, 19],
                    weight_scale=weight_scale,
                    reg=1e-5,
                    dtype=torch.float32,
                    device='cuda'
                    )

  solver = Solver(model, small_data,
                num_epochs=1, batch_size=128,
                update_rule=sgd_momentum,
                optim_config={
                  'learning_rate': 2e-3,
                },
                print_every=20, device='cuda')
  solver.train()
  solvers.append(solver)
```

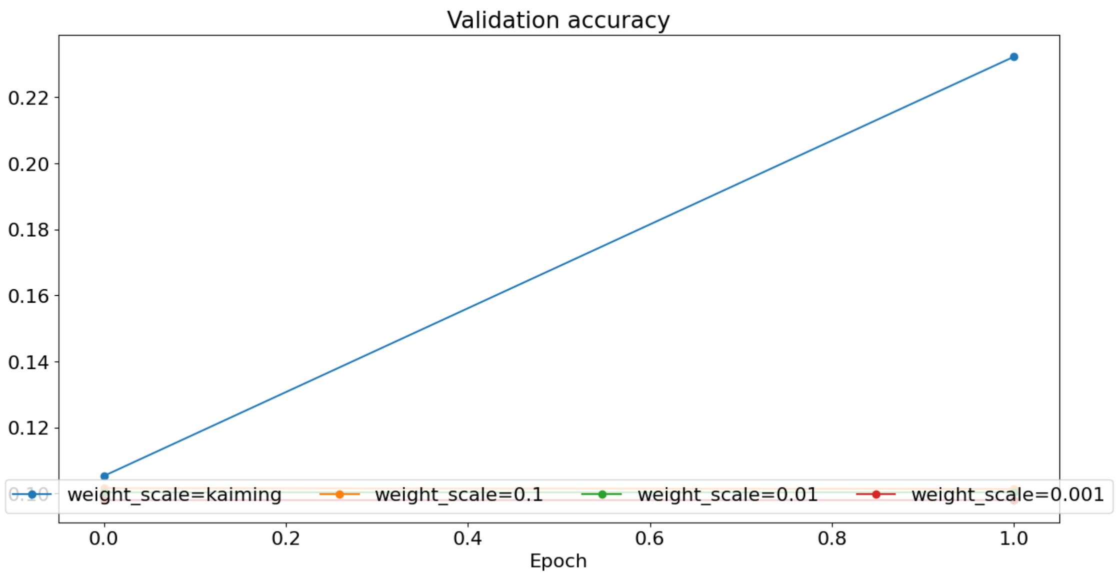
```
Solver with weight scale:  kaiming
(Time 0.08 sec; Iteration 1 / 78) loss: 2.330616
(Epoch 0 / 1) train acc: 0.109000; val_acc: 0.105300
(Time 3.20 sec; Iteration 21 / 78) loss: 2.320688
(Time 4.28 sec; Iteration 41 / 78) loss: 2.199364
(Time 5.36 sec; Iteration 61 / 78) loss: 2.197581
(Epoch 1 / 1) train acc: 0.227000; val_acc: 0.232300
Solver with weight scale:  0.1
(Time 0.06 sec; Iteration 1 / 78) loss: 112.550392
(Epoch 0 / 1) train acc: 0.105000; val_acc: 0.101600
(Time 3.07 sec; Iteration 21 / 78) loss: nan
(Time 4.13 sec; Iteration 41 / 78) loss: nan
(Time 5.21 sec; Iteration 61 / 78) loss: nan
(Epoch 1 / 1) train acc: 0.092000; val_acc: 0.101400
Solver with weight scale:  0.01
(Time 0.05 sec; Iteration 1 / 78) loss: 2.304122
```

```
(Epoch 0 / 1) train acc: 0.120000; val_acc: 0.100300
(Time 3.79 sec; Iteration 21 / 78) loss: 2.304196
(Time 5.05 sec; Iteration 41 / 78) loss: 2.303956
(Time 6.13 sec; Iteration 61 / 78) loss: 2.303877
(Epoch 1 / 1) train acc: 0.111000; val_acc: 0.100300
Solver with weight scale:  0.001
(Time 0.05 sec; Iteration 1 / 78) loss: 2.302599
(Epoch 0 / 1) train acc: 0.095000; val_acc: 0.098000
(Time 3.10 sec; Iteration 21 / 78) loss: 2.302582
(Time 4.15 sec; Iteration 41 / 78) loss: 2.302862
(Time 5.22 sec; Iteration 61 / 78) loss: 2.302748
(Epoch 1 / 1) train acc: 0.094000; val_acc: 0.098000
```

[ ]:
```
def plot_training_history_init(title, xlabel, solvers, labels, plot_fn,␣
 ↪marker='-o'):
  plt.title(title)
  plt.xlabel(xlabel)
  for solver, label in zip(solvers, labels):
    data = plot_fn(solver)
    label = 'weight_scale=' + str(label)
    plt.plot(data, marker, label=label)
  plt.legend(loc='lower center', ncol=len(solvers))

plt.subplot(3, 1, 1)
plot_training_history_init('Training loss','Iteration', solvers, weight_scales,
                            lambda x: x.loss_history, marker='o')
plt.subplot(3, 1, 2)
plot_training_history_init('Training accuracy','Epoch', solvers, weight_scales,
                           lambda x: x.train_acc_history)
plt.subplot(3, 1, 3)
plot_training_history_init('Validation accuracy','Epoch', solvers,␣
 ↪weight_scales,
                           lambda x: x.val_acc_history)
plt.gcf().set_size_inches(15, 25)
plt.show()
```


Training loss
1e7
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50
60
70
80
Iteration
weight_scale=kaiming
weight_scale=0.1
weight_scale=0.01
weight_scale=0.001


Training accuracy
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0.0
0.2
0.4
0.6
0.8
1.0
Epoch
weight_scale=kaiming
weight_scale=0.1
weight_scale=0.01
weight_scale=0.001


Validation accuracy
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0.0
0.2
0.4
0.6
0.8
1.0
Epoch
weight_scale=kaiming
weight_scale=0.1
weight_scale=0.01
weight_scale=0.001

12 Train a good model!

Train the best convolutional model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 71% accuracy on the validation set using a convolutional net, within 60 seconds of training.

You might find it useful to use batch normalization in your model. However, since we do not ask you to implement it CUDA-friendly, it might slow down training.

Implement `create_convolutional_solver_instance` while making sure to use the initialize your model with the input `dtype` and `device`, as well as initializing the solver on the input `device`.

Hint: Your model does not have to be too deep.

Hint 2: We used `batch_size = 128` for training a model with 74% validation accuracy. You don't have to follow this, but it would save your time for hyperparameter search.

Hint 3: Note that we import all the functions from fully_connected_networks, so feel free to use the optimizers you've already imolemented; e.g., adam.

[ ]:
```python
from convolutional_networks import DeepConvNet,␣
 ↪create_convolutional_solver_instance

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = True

solver = create_convolutional_solver_instance(data_dict, torch.float32, "cuda")

solver.train(time_limit=60)

torch.backends.cudnn.benchmark = False
```

```
(Time 0.01 sec; Iteration 1 / 1330) loss: 2.611783
(Epoch 0 / 5) train acc: 0.140000; val_acc: 0.113000
(Time 0.68 sec; Iteration 51 / 1330) loss: 1.853167
(Time 1.13 sec; Iteration 101 / 1330) loss: 1.471103
(Time 1.58 sec; Iteration 151 / 1330) loss: 1.492934
(Time 2.04 sec; Iteration 201 / 1330) loss: 1.404246
(Time 2.51 sec; Iteration 251 / 1330) loss: 1.486559
(Epoch 1 / 5) train acc: 0.619000; val_acc: 0.595000
(Time 3.20 sec; Iteration 301 / 1330) loss: 1.292696
(Time 3.65 sec; Iteration 351 / 1330) loss: 1.309885
(Time 4.11 sec; Iteration 401 / 1330) loss: 1.065599
(Time 4.56 sec; Iteration 451 / 1330) loss: 1.111165
(Time 5.02 sec; Iteration 501 / 1330) loss: 1.284971
(Epoch 2 / 5) train acc: 0.690000; val_acc: 0.662800
(Time 5.66 sec; Iteration 551 / 1330) loss: 1.179495
```

```
(Time 6.12 sec; Iteration 601 / 1330) loss: 1.000875
(Time 6.57 sec; Iteration 651 / 1330) loss: 1.102147
(Time 7.03 sec; Iteration 701 / 1330) loss: 0.967604
(Time 7.48 sec; Iteration 751 / 1330) loss: 0.793440
(Epoch 3 / 5) train acc: 0.743000; val_acc: 0.692200
(Time 8.13 sec; Iteration 801 / 1330) loss: 0.862282
(Time 8.58 sec; Iteration 851 / 1330) loss: 0.957499
(Time 9.10 sec; Iteration 901 / 1330) loss: 0.935063
(Time 9.63 sec; Iteration 951 / 1330) loss: 0.955283
(Time 10.18 sec; Iteration 1001 / 1330) loss: 0.962701
(Time 10.71 sec; Iteration 1051 / 1330) loss: 0.962543
(Epoch 4 / 5) train acc: 0.775000; val_acc: 0.702000
(Time 11.57 sec; Iteration 1101 / 1330) loss: 1.014470
(Time 12.16 sec; Iteration 1151 / 1330) loss: 0.979079
(Time 12.62 sec; Iteration 1201 / 1330) loss: 0.951437
(Time 13.08 sec; Iteration 1251 / 1330) loss: 0.830947
(Time 13.54 sec; Iteration 1301 / 1330) loss: 0.711689
(Epoch 5 / 5) train acc: 0.768000; val_acc: 0.712700
```

13 Test your model!

Run your best model on the validation and test sets. You should achieve above 71% accuracy on the validation set and 70% accuracy on the test set.

(Our best model gets 74.3% validation accuracy and 73.5% test accuracy – can you beat ours?)

```
[ ]: print('Validation set accuracy: ', solver.check_accuracy(data_dict['X_val'],␣
      ↪data_dict['y_val']))
     print('Test set accuracy: ', solver.check_accuracy(data_dict['X_test'],␣
      ↪data_dict['y_test']))
```

```
Validation set accuracy:  0.7127000093460083
Test set accuracy:  0.7055999636650085
```

If you're happy with the model's perfromance, run the following cell to save it.

We will also reload the model and run it on the training data to verify it's the right weights.

```
[ ]: path = os.path.join(GOOGLE_DRIVE_PATH, 'one_minute_deepconvnet.pth')
     solver.model.save(path)

     # Create a new instance
     from convolutional_networks import DeepConvNet,␣
      ↪create_convolutional_solver_instance

     solver = create_convolutional_solver_instance(data_dict, torch.float32, "cuda")

     # Load model
     solver.model.load(path, dtype=torch.float32, device='cuda')
```

```
# Evaluate on validation set
print('Validation set accuracy: ', solver.check_accuracy(data_dict['X_val'],␣
 ↪data_dict['y_val']))
print('Test set accuracy: ', solver.check_accuracy(data_dict['X_test'],␣
 ↪data_dict['y_test']))
```

```
Saved in drive/My Drive/ITE4052/A4/one_minute_deepconvnet.pth
load checkpoint file: drive/My Drive/ITE4052/A4/one_minute_deepconvnet.pth
Validation set accuracy:  0.7127000093460083
Test set accuracy:  0.7055999636650085
```

14 Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization which was proposed by [1] in 2015.

The idea is relatively straightforward. Machine learning methods tend to work better when their input data consists of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features; this will ensure that the first layer of the network sees data that follows a nice distribution. However, even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance since they are output from earlier layers in the network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors of [1] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [1] proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension.

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

14.1 Batch normalization: forward

Implement the batch normalization forward pass in the function `BatchNorm.forward`. Once you have done so, run the following to test your implementation.

Referencing the paper linked to above in [1] may be helpful!

After implementing the forward pass for batch normalization, you can run the following to sanity check your implementation. After running batch normalization with beta=0 and gamma=1, the data should have zero mean and unit variance.

After running batch normalization with nontrivial beta and gamma, the output data should have mean approximately equal to beta, and std approximatly equal to gamma.

[ ]:
```python
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization
from convolutional_networks import BatchNorm

def print_mean_std(x,dim=0):
  means = ['%.3f' % xx for xx in x.mean(dim=dim).tolist()]
  stds = ['%.3f' % xx for xx in x.std(dim=dim).tolist()]
  print('  means: ', means)
  print('  stds:  ', stds)
  print()

# Simulate the forward pass for a two-layer network
reset_seed(0)
N, D1, D2, D3 = 200, 50, 60, 3
X = torch.randn(N, D1, dtype=torch.float64, device='cuda')
W1 = torch.randn(D1, D2, dtype=torch.float64, device='cuda')
W2 = torch.randn(D2, D3, dtype=torch.float64, device='cuda')
a = X.matmul(W1).clamp(min=0.).matmul(W2)

print('Before batch normalization:')
print_mean_std(a,dim=0)

# Run with gamma=1, beta=0. Means should be close to zero and stds close to one
gamma = torch.ones(D3, dtype=torch.float64, device='cuda')
beta = torch.zeros(D3, dtype=torch.float64, device='cuda')
print('After batch normalization (gamma=1, beta=0)')
a_norm, _ = BatchNorm.forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,dim=0)

# Run again with nontrivial gamma and beta. Now means should be close to beta
# and std should be close to gamma.
gamma = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float64, device='cuda')
beta = torch.tensor([11.0, 12.0, 13.0], dtype=torch.float64, device='cuda')
print('After batch normalization (gamma=', gamma.tolist(), ', beta=', beta.
 ↪tolist(), ')')
a_norm, _ = BatchNorm.forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,dim=0)
```

```
Before batch normalization:
  means:  ['52.046', '11.122', '10.243']
  stds:    ['34.646', '30.732', '39.429']

After batch normalization (gamma=1, beta=0)
  means:  ['-0.000', '-0.000', '0.000']
  stds:   ['1.003', '1.003', '1.003']

After batch normalization (gamma= [1.0, 2.0, 3.0] , beta= [11.0, 12.0, 13.0] )
  means:  ['11.000', '12.000', '13.000']
  stds:   ['1.003', '2.005', '3.008']
```

We can sanity-check the test-time forward pass of batch normalization by running the following. First we run the training-time forward pass many times to “warm up” the running averages. If we then run a test-time forward pass, the output should have approximately zero mean and unit variance.

```
[ ]: from convolutional_networks import BatchNorm

reset_seed(0)
N, D1, D2, D3 = 200, 50, 60, 3
W1 = torch.randn(D1, D2, dtype=torch.float64, device='cuda')
W2 = torch.randn(D2, D3, dtype=torch.float64, device='cuda')

bn_param = {'mode': 'train'}
gamma = torch.ones(D3, dtype=torch.float64, device='cuda')
beta = torch.zeros(D3, dtype=torch.float64, device='cuda')

for t in range(500):
  X = torch.randn(N, D1, dtype=torch.float64, device='cuda')
  a = X.matmul(W1).clamp(min=0.).matmul(W2)
  BatchNorm.forward(a, gamma, beta, bn_param)

bn_param['mode'] = 'test'
X = torch.randn(N, D1, dtype=torch.float64, device='cuda')
a = X.matmul(W1).clamp(min=0.).matmul(W2)
a_norm, _ = BatchNorm.forward(a, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After batch normalization (test-time):')
print_mean_std(a_norm,dim=0)
```

```
After batch normalization (test-time):
  means:  ['0.031', '-0.051', '0.061']
  stds:   ['1.011', '0.958', '1.068']
```

14.2 Batch normalization: backward

Now implement the backward pass for batch normalization in the function `BatchNorm.backward`.

To derive the backward pass you should write out the computation graph for batch normalization and backprop through each of the intermediate nodes. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.

Please don't forget to implement the train and test mode separately.

Once you have finished, run the following to numerically check your backward pass.

[ ]:
```python
from convolutional_networks import BatchNorm

# Gradient check batchnorm backward pass
reset_seed(0)
N, D = 4, 5
x = 5 * torch.randn(N, D, dtype=torch.float64, device='cuda') + 12
gamma = torch.randn(D, dtype=torch.float64, device='cuda')
beta = torch.randn(D, dtype=torch.float64, device='cuda')
dout = torch.randn(N, D, dtype=torch.float64, device='cuda')

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

dx_num = ite4052.grad.compute_numeric_gradient(fx, x, dout)
da_num = ite4052.grad.compute_numeric_gradient(fg, gamma.clone(), dout)
db_num = ite4052.grad.compute_numeric_gradient(fb, beta.clone(), dout)

_, cache = BatchNorm.forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = BatchNorm.backward(dout, cache)
# You should expect to see relative errors between 1e-12 and 1e-9
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
print('dgamma error: ', ite4052.grad.rel_error(da_num, dgamma))
print('dbeta error: ', ite4052.grad.rel_error(db_num, dbeta))
```

```
dx error:  9.592374299081715e-09
dgamma error:  4.765758100649042e-10
dbeta error:  2.919336030277982e-10
```

14.3 (Challenge Question) Batch Normalization: alternative backward

In class we talked about two different implementations for the sigmoid backward pass. One strategy is to write out a computation graph composed of simple operations and backprop through all intermediate values. Another strategy is to work out the derivatives on paper. For example, you can derive a very simple formula for the sigmoid function's backward pass by simplifying gradients on paper.

Surprisingly, it turns out that you can do a similar simplification for the batch normalization backward pass too!

In the forward pass, given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}$,

we first calculate the mean μ and variance v. With μ and v calculated, we can calculate the standard deviation σ and normalized data Y. The equations and graph illustration below describe the computation (y_i is the i-th element of the vector Y).

$$\mu = \frac{1}{N}\sum_{k=1}^{N} x_k \qquad v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2 \tag{1}$$

$$\sigma = \sqrt{v + \epsilon} \qquad y_i = \frac{x_i - \mu}{\sigma} \tag{2}$$

The meat of our problem during backpropagation is to compute $\frac{\partial L}{\partial X}$, given the upstream gradient we receive, $\frac{\partial L}{\partial Y}$. To do this, recall the chain rule in calculus gives us $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X}$.

The unknown/hart part is $\frac{\partial Y}{\partial X}$. We can find this by first deriving step-by-step our local gradients at $\frac{\partial v}{\partial X}$, $\frac{\partial \mu}{\partial X}$, $\frac{\partial \sigma}{\partial v}$, $\frac{\partial Y}{\partial \sigma}$, and $\frac{\partial Y}{\partial \mu}$, and then use the chain rule to compose these gradients (which appear in the form of vectors!) appropriately to compute $\frac{\partial Y}{\partial X}$.

If it's challenging to directly reason about the gradients over X and Y which require matrix multiplication, try reasoning about the gradients in terms of individual elements x_i and y_i first: in that case, you will need to come up with the derivations for $\frac{\partial L}{\partial x_i}$, by relying on the Chain Rule to first calculate the intermediate $\frac{\partial \mu}{\partial x_i}$, $\frac{\partial v}{\partial x_i}$, $\frac{\partial \sigma}{\partial x_i}$, then assemble these pieces to calculate $\frac{\partial y_i}{\partial x_i}$.

You should make sure each of the intermediary gradient derivations are all as simplified as possible, for ease of implementation.

After doing so, implement the simplified batch normalization backward pass in the function `BatchNorm.backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

[ ]:
```python
from convolutional_networks import BatchNorm

reset_seed(0)
N, D = 128, 2048
x = 5 * torch.randn(N, D, dtype=torch.float64, device='cuda') + 12
gamma = torch.randn(D, dtype=torch.float64, device='cuda')
beta = torch.randn(D, dtype=torch.float64, device='cuda')
dout = torch.randn(N, D, dtype=torch.float64, device='cuda')

bn_param = {'mode': 'train'}
out, cache = BatchNorm.forward(x, gamma, beta, bn_param)
```

```
t1 = time.time()
dx1, dgamma1, dbeta1 = BatchNorm.backward(dout, cache)
t2 = time.time()
dx2, dgamma2, dbeta2 = BatchNorm.backward_alt(dout, cache)
t3 = time.time()

print('dx difference: ', ite4052.grad.rel_error(dx1, dx2))
print('dgamma difference: ', ite4052.grad.rel_error(dgamma1, dgamma2))
print('dbeta difference: ', ite4052.grad.rel_error(dbeta1, dbeta2))
print('speedup: %.2fx' % ((t2 - t1) / (t3 - t2)))
```

```
dx difference:  1.6247876458830077e-16
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.11x
```

15 Spatial Batch Normalization

As proposed in the original paper, batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

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 needs to accept inputs of shape `(N, C, H, W)` and produce 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.

If the feature map was produced using convolutions, then we expect every feature channel's statistics e.g. mean, variance to be relatively consistent both between different images, and different locations within the same image – after all, every feature channel is produced by the same convolutional filter! Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over the minibatch dimension `N` as well the spatial dimensions `H` and `W`.

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

15.1 Spatial batch normalization: forward

Implement the forward pass for spatial batch normalization in the function `SpatialBatchNorm.forward`. Check your implementation by running the following:

After implementing the forward pass for spatial batch normalization, you can run the following to sanity check your code.

```
[ ]: from convolutional_networks import SpatialBatchNorm

reset_seed(0)
# 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 * torch.randn(N, C, H, W, dtype=torch.float64, device='cuda') + 10

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

# Means should be close to zero and stds close to one
gamma = torch.ones(C, dtype=torch.float64, device='cuda')
beta = torch.zeros(C,dtype=torch.float64, device='cuda')
bn_param = {'mode': 'train'}
out, _ = SpatialBatchNorm.forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(dim=(0, 2, 3)))
print('  Stds: ', out.std(dim=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma = torch.tensor([3, 4, 5], dtype=torch.float64, device='cuda')
beta = torch.tensor([6, 7, 8], dtype=torch.float64, device='cuda')
out, _ = SpatialBatchNorm.forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(dim=(0, 2, 3)))
print('  Stds: ', out.std(dim=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  torch.Size([2, 3, 4, 5])
  Means:  tensor([ 9.5501, 10.2173,  9.8379], device='cuda:0',
dtype=torch.float64)
  Stds:  tensor([3.9212, 4.7963, 3.6011], device='cuda:0', dtype=torch.float64)
After spatial batch normalization:
  Shape:  torch.Size([2, 3, 4, 5])
  Means:  tensor([-2.2204e-16,  1.9706e-16, -2.1094e-16], device='cuda:0',
       dtype=torch.float64)
  Stds:  tensor([1.0127, 1.0127, 1.0127], device='cuda:0', dtype=torch.float64)
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  torch.Size([2, 3, 4, 5])
  Means:  tensor([6., 7., 8.], device='cuda:0', dtype=torch.float64)
  Stds:  tensor([3.0382, 4.0510, 5.0637], device='cuda:0', dtype=torch.float64)
```

Similar to the vanilla batch normalization implementation, run the following to sanity-check the test-time forward pass of spatial batch normalization.

[ ]:
```
reset_seed(0)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = torch.ones(C, dtype=torch.float64, device='cuda')
beta = torch.zeros(C, dtype=torch.float64, device='cuda')
for t in range(50):
  x = 2.3 * torch.randn(N, C, H, W, dtype=torch.float64, device='cuda') + 13
  SpatialBatchNorm.forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * torch.randn(N, C, H, W, dtype=torch.float64, device='cuda') + 13
a_norm, _ = SpatialBatchNorm.forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(dim=(0, 2, 3)))
print('  stds: ', a_norm.std(dim=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  tensor([0.0188, 0.0145, 0.0422, 0.0231], device='cuda:0',
dtype=torch.float64)
  stds:  tensor([0.9861, 1.0143, 1.0138, 0.9916], device='cuda:0',
dtype=torch.float64)
```

15.2 Spatial batch normalization: backward

Implement the backward pass for spatial batch normalization in the function `SpatialBatchNorm.backward`.

After implementing the backward pass for spatial batch normalization, run the following to perform numeric gradient checking on your implementation. You should see errors less than `1e-6`.

[ ]:
```
reset_seed(0)
N, C, H, W = 2, 3, 4, 5
x = 5 * torch.randn(N, C, H, W, dtype=torch.float64, device='cuda') + 12
gamma = torch.randn(C, dtype=torch.float64, device='cuda')
beta = torch.randn(C, dtype=torch.float64, device='cuda')
dout = torch.randn(N, C, H, W, dtype=torch.float64, device='cuda')

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

```
dx_num = ite4052.grad.compute_numeric_gradient(fx, x, dout)
da_num = ite4052.grad.compute_numeric_gradient(fg, gamma, dout)
db_num = ite4052.grad.compute_numeric_gradient(fb, beta, dout)

_, cache = SpatialBatchNorm.forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = SpatialBatchNorm.backward(dout, cache)
print('dx error: ', ite4052.grad.rel_error(dx_num, dx))
print('dgamma error: ', ite4052.grad.rel_error(da_num, dgamma))
print('dbeta error: ', ite4052.grad.rel_error(db_num, dbeta))
```

```
dx error:  1.8020575578408637e-08
dgamma error:  2.8823463766472374e-10
dbeta error:  2.631779170090561e-10
```

16 “Sandwich” layers with batch normalization

Again, below you will find sandwich layers that implement a few commonly used patterns for convolutional networks. We include the functions in `convolutional_networks.py` but you can see them here for your convenience.

```
class Linear_BatchNorm_ReLU(object):

  @staticmethod
  def forward(x, w, b, gamma, beta, bn_param):
    """
    Convenience layer that performs an linear transform, batch normalization,
    and ReLU.
    Inputs:
    - x: Array of shape (N, D1); input to the linear layer
    - w, b: Arrays of shape (D2, D2) and (D2,) giving the weight and bias for
      the linear transform.
    - gamma, beta: Arrays of shape (D2,) and (D2,) giving scale and shift
      parameters for batch normalization.
    - bn_param: Dictionary of parameters for batch normalization.
    Returns:
    - out: Output from ReLU, of shape (N, D2)
    - cache: Object to give to the backward pass.
    """
    a, fc_cache = Linear.forward(x, w, b)
    a_bn, bn_cache = BatchNorm.forward(a, gamma, beta, bn_param)
    out, relu_cache = ReLU.forward(a_bn)
    cache = (fc_cache, bn_cache, relu_cache)
    return out, cache

  @staticmethod
  def backward(dout, cache):
    """
    Backward pass for the linear-batchnorm-relu convenience layer.
    """
    fc_cache, bn_cache, relu_cache = cache
    da_bn = ReLU.backward(dout, relu_cache)
    da, dgamma, dbeta = BatchNorm.backward(da_bn, bn_cache)
    dx, dw, db = Linear.backward(da, fc_cache)
    return dx, dw, db, dgamma, dbeta


class Conv_BatchNorm_ReLU(object):

  @staticmethod
  def forward(x, w, b, gamma, beta, conv_param, bn_param):
    a, conv_cache = FastConv.forward(x, w, b, conv_param)
    an, bn_cache = SpatialBatchNorm.forward(a, gamma, beta, bn_param)
    out, relu_cache = ReLU.forward(an)
    cache = (conv_cache, bn_cache, relu_cache)
    return out, cache

  @staticmethod
  def backward(dout, cache):
    conv_cache, bn_cache, relu_cache = cache
    dan = ReLU.backward(dout, relu_cache)
    da, dgamma, dbeta = SpatialBatchNorm.backward(dan, bn_cache)
    dx, dw, db = FastConv.backward(da, conv_cache)
    return dx, dw, db, dgamma, dbeta


class Conv_BatchNorm_ReLU_Pool(object):

  @staticmethod
  def forward(x, w, b, gamma, beta, conv_param, bn_param, pool_param):
    a, conv_cache = FastConv.forward(x, w, b, conv_param)
    an, bn_cache = SpatialBatchNorm.forward(a, gamma, beta, bn_param)
    s, relu_cache = ReLU.forward(an)
    out, pool_cache = FastMaxPool.forward(s, pool_param)
    cache = (conv_cache, bn_cache, relu_cache, pool_cache)
    return out, cache

  @staticmethod
  def backward(dout, cache):
    conv_cache, bn_cache, relu_cache, pool_cache = cache
    ds = FastMaxPool.backward(dout, pool_cache)
    dan = ReLU.backward(ds, relu_cache)
    da, dgamma, dbeta = SpatialBatchNorm.backward(dan, bn_cache)
    dx, dw, db = FastConv.backward(da, conv_cache)
    return dx, dw, db, dgamma, dbeta
```

17 Convolutional nets with batch normalization

Now that you have a working implementation for batch normalization, go back to your DeepConvnet. Modify your implementation to add batch normalization.

Concretely, when the `batchnorm` flag is set to `True` in the constructor, you should insert a batch normalization layer before each ReLU nonlinearity. The outputs from the last linear layer of the network should not be normalized. Once you are done, run the following to gradient-check your implementation.

In the reg=0 case, you should see errors less than `1e-6` for all weights and batchnorm parameters (beta and gamma); for biases you will see high relative errors due to the extremely small magnitude of both numeric and analytic gradients.

In the reg=3.14 case, you should see errors less than `1e-6` for all parameters.

```
[ ]: from convolutional_networks import DeepConvNet
     reset_seed(0)

     num_inputs = 2
     input_dims = (3, 8, 8)
     num_classes = 10
     X = torch.randn(num_inputs, *input_dims, dtype=torch.float64, device='cuda')
     y = torch.randint(num_classes, size=(num_inputs,), dtype=torch.int64,␣
      ↪device='cuda')

     for reg in [0, 3.14]:
       print('Running check with reg = ', reg)
       model = DeepConvNet(input_dims=input_dims, num_classes=num_classes,
                           num_filters=[8, 8, 8],
                           max_pools=[0, 2],
                           reg=reg, batchnorm=True,
                           weight_scale='kaiming',
                           dtype=torch.float64, device='cuda')

       loss, grads = model.loss(X, y)
       # The relative errors should be up to the order of e-3
       for name in sorted(grads):
         f = lambda _: model.loss(X, y)[0]
         grad_num = ite4052.grad.compute_numeric_gradient(f, model.params[name])
         print('%s max relative error: %e' % (name, ite4052.grad.rel_error(grad_num,␣
      ↪grads[name])))
       print()
```

```
Running check with reg =  0
W1 max relative error: 5.471040e-09
W2 max relative error: 3.761870e-09
W3 max relative error: 3.613076e-09
W4 max relative error: 1.222156e-09
```

```
b1 max relative error: 1.000000e+00
b2 max relative error: 1.000000e+00
b3 max relative error: 1.000000e+00
b4 max relative error: 3.343807e-09
beta1 max relative error: 8.554469e-09
beta2 max relative error: 1.191723e-08
beta3 max relative error: 1.260690e-09
gamma1 max relative error: 6.053368e-09
gamma2 max relative error: 9.161435e-09
gamma3 max relative error: 1.550969e-09

Running check with reg =  3.14
W1 max relative error: 1.391388e-08
W2 max relative error: 2.422010e-08
W3 max relative error: 4.043007e-08
W4 max relative error: 3.237231e-08
b1 max relative error: 1.526557e-06
b2 max relative error: 8.326673e-07
b3 max relative error: 3.469447e-07
b4 max relative error: 8.585677e-08
beta1 max relative error: 1.342468e-07
beta2 max relative error: 9.276992e-08
beta3 max relative error: 1.082601e-07
gamma1 max relative error: 8.938120e-08
gamma2 max relative error: 1.085495e-07
gamma3 max relative error: 5.679801e-08
```

18 Batchnorm for deep convolutional networks

Run the following to train a deep convolutional network on a subset of 500 training examples both with and without batch normalization.

[ ]:
```python
from convolutional_networks import DeepConvNet
reset_seed(0)

# Try training a deep convolutional net with batchnorm
num_train = 500
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}
input_dims = data_dict['X_train'].shape[1:]

bn_model = DeepConvNet(input_dims=input_dims, num_classes=10,
                        num_filters=[16, 32, 32, 64, 64],
                        max_pools=[0, 1, 2, 3, 4],
                        weight_scale='kaiming',
                        batchnorm=True,
                        reg=1e-5,  dtype=torch.float32, device='cuda')
model = DeepConvNet(input_dims=input_dims, num_classes=10,
                    num_filters=[16, 32, 32, 64, 64],
                    max_pools=[0, 1, 2, 3, 4],
                    weight_scale='kaiming',
                    batchnorm=False,
                    reg=1e-5,  dtype=torch.float32, device='cuda')

print('Solver with batch norm:')
bn_solver = Solver(bn_model, small_data,
                   num_epochs=10, batch_size=100,
                   update_rule=adam,
                   optim_config={
                     'learning_rate': 1e-3,
                   },
                   print_every=20, device='cuda')
bn_solver.train()

print('\nSolver without batch norm:')
solver = Solver(model, small_data,
                num_epochs=10, batch_size=100,
                update_rule=adam,
                optim_config={
                  'learning_rate': 1e-3,
                },
                print_every=20, device='cuda')
solver.train()
```

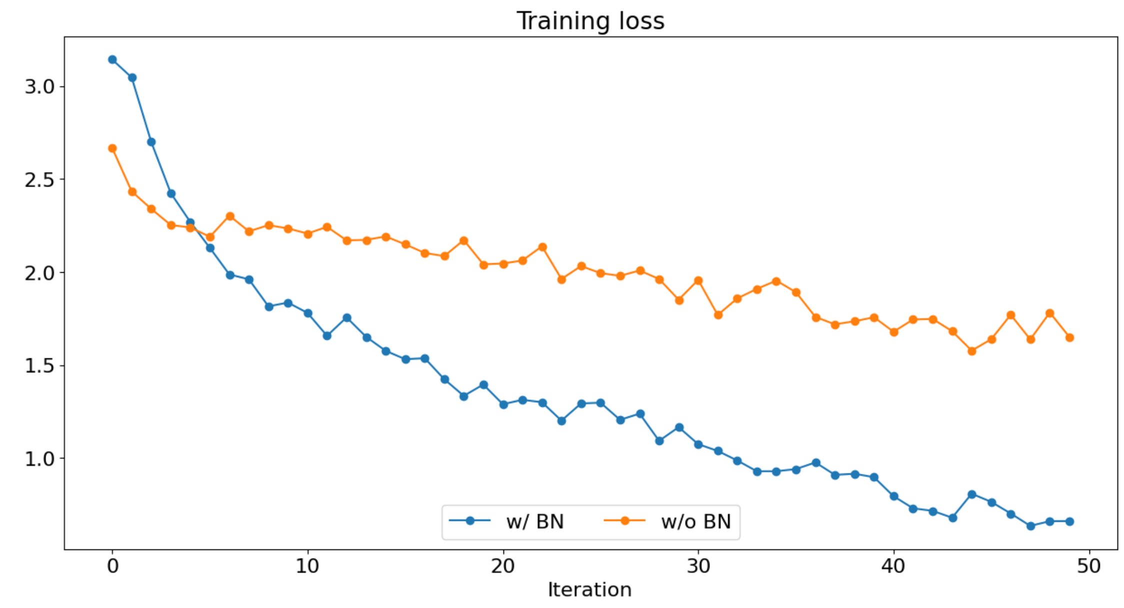
```
Solver with batch norm:
(Time 0.03 sec; Iteration 1 / 50) loss: 3.142073
(Epoch 0 / 10) train acc: 0.098000; val_acc: 0.117100
(Epoch 1 / 10) train acc: 0.152000; val_acc: 0.120600
(Epoch 2 / 10) train acc: 0.144000; val_acc: 0.114400
(Epoch 3 / 10) train acc: 0.184000; val_acc: 0.146200
(Epoch 4 / 10) train acc: 0.464000; val_acc: 0.268200
(Time 2.67 sec; Iteration 21 / 50) loss: 1.289564
(Epoch 5 / 10) train acc: 0.620000; val_acc: 0.318000
(Epoch 6 / 10) train acc: 0.736000; val_acc: 0.327900
(Epoch 7 / 10) train acc: 0.770000; val_acc: 0.323100
(Epoch 8 / 10) train acc: 0.824000; val_acc: 0.334700
(Time 5.95 sec; Iteration 41 / 50) loss: 0.795251
(Epoch 9 / 10) train acc: 0.890000; val_acc: 0.351500
(Epoch 10 / 10) train acc: 0.912000; val_acc: 0.358600
```

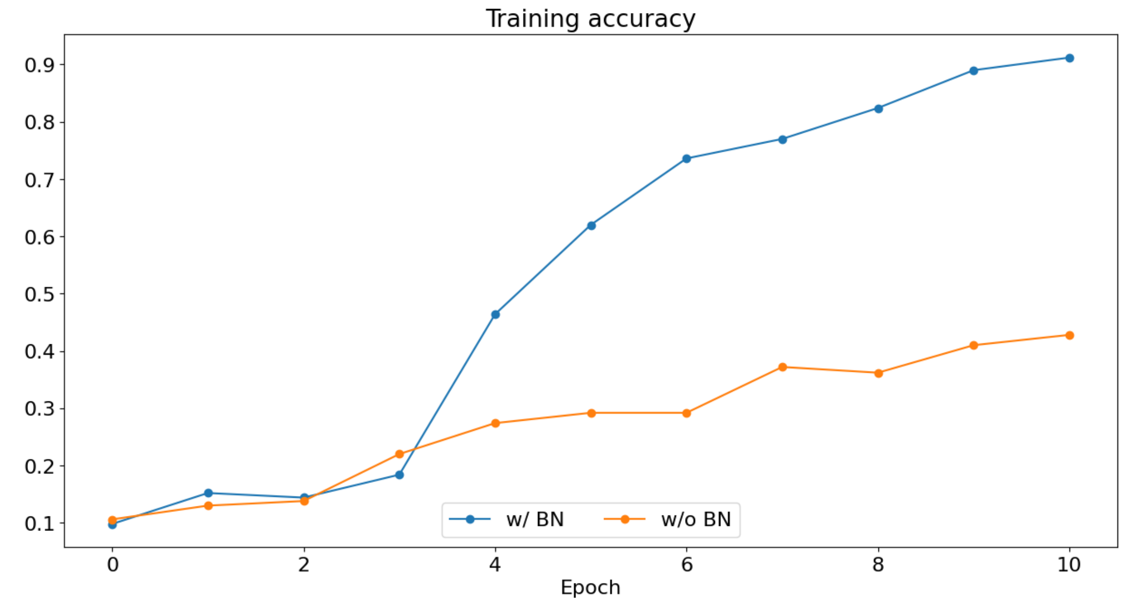
```
Solver without batch norm:
(Time 0.02 sec; Iteration 1 / 50) loss: 2.666687
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.099200
(Epoch 1 / 10) train acc: 0.130000; val_acc: 0.101900
(Epoch 2 / 10) train acc: 0.138000; val_acc: 0.106500
(Epoch 3 / 10) train acc: 0.220000; val_acc: 0.153000
(Epoch 4 / 10) train acc: 0.274000; val_acc: 0.219900
(Time 1.52 sec; Iteration 21 / 50) loss: 2.045820
(Epoch 5 / 10) train acc: 0.292000; val_acc: 0.224000
(Epoch 6 / 10) train acc: 0.292000; val_acc: 0.230700
(Epoch 7 / 10) train acc: 0.372000; val_acc: 0.266800
(Epoch 8 / 10) train acc: 0.362000; val_acc: 0.250400
(Time 2.78 sec; Iteration 41 / 50) loss: 1.679068
(Epoch 9 / 10) train acc: 0.410000; val_acc: 0.284000
(Epoch 10 / 10) train acc: 0.428000; val_acc: 0.279500
```

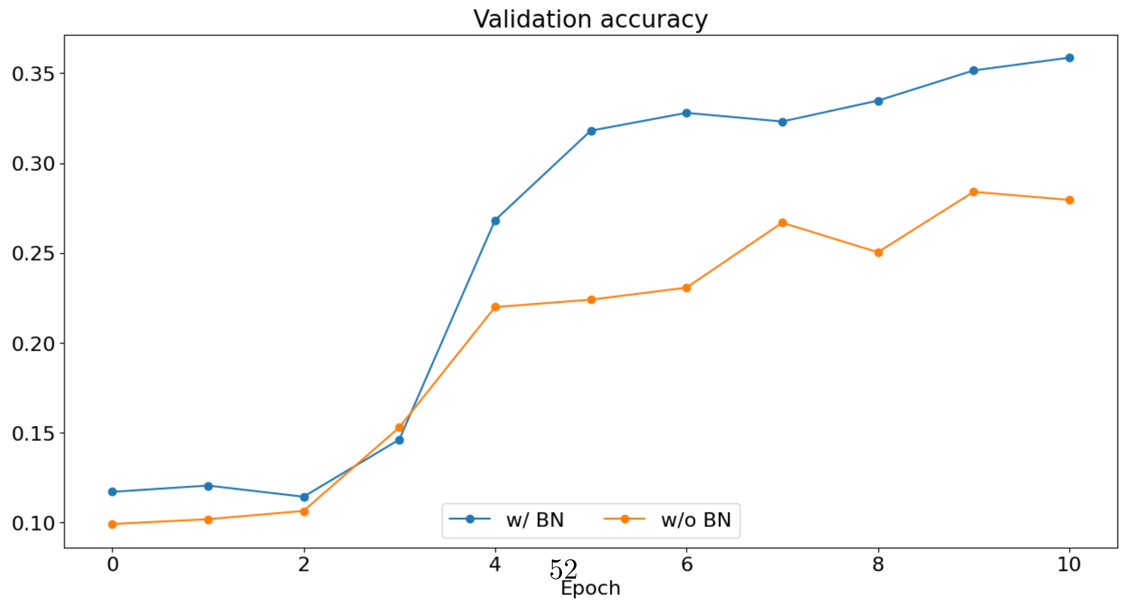
Run the following to visualize the results from two networks trained above. You should find that using batch normalization helps the network to converge much faster.

```
[ ]: def plot_training_history_bn(title, label, solvers, bn_solvers, plot_fn,␣
      ↪bl_marker='.', bn_marker='.', labels=None):
       """utility function for plotting training history"""
       plt.title(title)
       plt.xlabel(label)
       bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
       bl_plots = [plot_fn(solver) for solver in solvers]
       num_bn = len(bn_plots)
       num_bl = len(bl_plots)
       for i in range(num_bn):
         label='w/ BN'
         if labels is not None:
           label += str(labels[i])
         plt.plot(bn_plots[i], bn_marker, label=label)
       for i in range(num_bl):
         label='w/o BN'
         if labels is not None:
           label += str(labels[i])
         plt.plot(bl_plots[i], bl_marker, label=label)
       plt.legend(loc='lower center', ncol=num_bn+num_bl)

     plt.subplot(3, 1, 1)
     plot_training_history_bn('Training loss','Iteration', [solver], [bn_solver], \
                           lambda x: x.loss_history, bl_marker='-o', bn_marker='-o')
     plt.subplot(3, 1, 2)
     plot_training_history_bn('Training accuracy','Epoch', [solver], [bn_solver], \
                           lambda x: x.train_acc_history, bl_marker='-o',␣
      ↪bn_marker='-o')
```

```
plt.subplot(3, 1, 3)
plot_training_history_bn('Validation accuracy','Epoch', [solver], [bn_solver], \
                         lambda x: x.val_acc_history, bl_marker='-o',␣
 ↪bn_marker='-o')

plt.gcf().set_size_inches(15, 25)
plt.show()
```


Training loss
3.0
2.5
2.0
1.5
1.0
0
10
20
30
40
50
Iteration
w/ BN
w/o BN


Training accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
Epoch
w/ BN
w/o BN


Validation accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0
2
4
6
8
10
Epoch
w/ BN
w/o BN

19 Batch normalization and learning rate

We will now run a small experiment to study the interaction of batch normalization and learning rate.

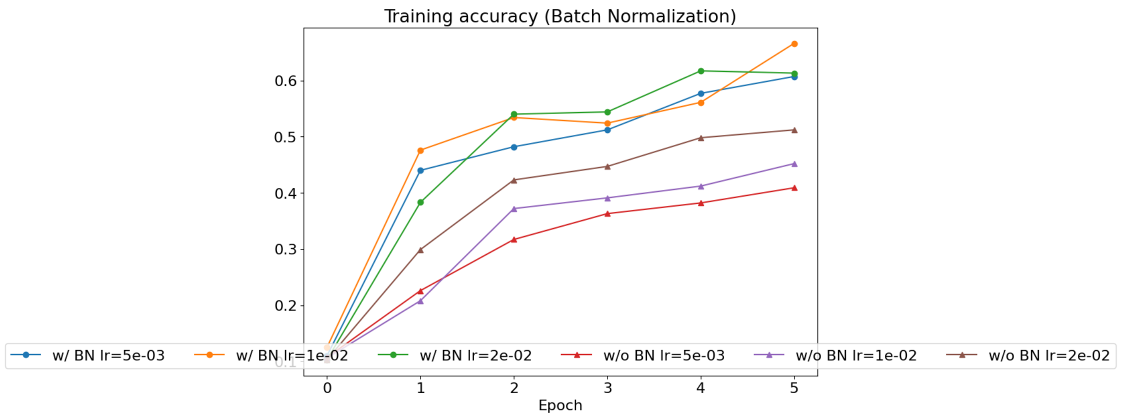
The first cell will train convolutional networks with different learning rates. The second layer will plot training accuracy and validation set accuracy over time. You should find that using batch normalization helps the network to be less dependent to the learning rate.

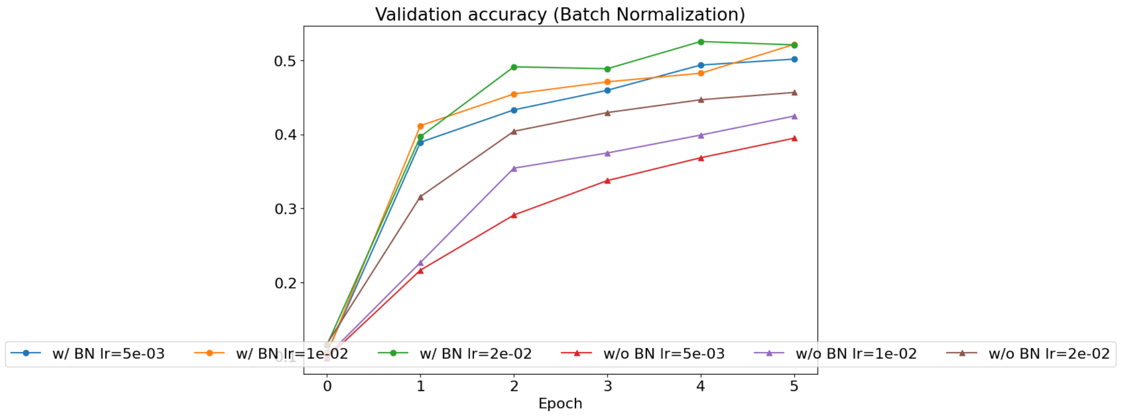
[ ]:
```
from convolutional_networks import DeepConvNet
from fully_connected_networks import sgd_momentum
reset_seed(0)

# Try training a very deep net with batchnorm
num_train = 10000
small_data = {
  'X_train': data_dict['X_train'][:num_train],
  'y_train': data_dict['y_train'][:num_train],
  'X_val': data_dict['X_val'],
  'y_val': data_dict['y_val'],
}
input_dims = data_dict['X_train'].shape[1:]
num_epochs = 5
lrs = [2e-1, 1e-1, 5e-2]
lrs = [5e-3, 1e-2, 2e-2]

solvers = []
for lr in lrs:
  print('No normalization: learning rate = ', lr)
  model = DeepConvNet(input_dims=input_dims, num_classes=10,
                      num_filters=[8, 8, 8],
                      max_pools=[0, 1, 2],
                      weight_scale='kaiming',
                      batchnorm=False,
                      reg=1e-5, dtype=torch.float32, device='cuda')
  solver = Solver(model, small_data,
                  num_epochs=num_epochs, batch_size=100,
                  update_rule=sgd_momentum,
                  optim_config={
                    'learning_rate': lr,
                  },
                  verbose=False, device='cuda')
  solver.train()
  solvers.append(solver)
```

```
bn_solvers = []
for lr in lrs:
  print('Normalization: learning rate = ', lr)
  bn_model = DeepConvNet(input_dims=input_dims, num_classes=10,
                       num_filters=[8, 8, 16, 16, 32, 32],
                       max_pools=[1, 3, 5],
                       weight_scale='kaiming',
                       batchnorm=True,
                       reg=1e-5, dtype=torch.float32, device='cuda')
  bn_solver = Solver(bn_model, small_data,
                     num_epochs=num_epochs, batch_size=128,
                     update_rule=sgd_momentum,
                     optim_config={
                       'learning_rate': lr,
                     },
                     verbose=False, device='cuda')
  bn_solver.train()
  bn_solvers.append(bn_solver)
```

```
No normalization: learning rate =  0.005
No normalization: learning rate =  0.01
No normalization: learning rate =  0.02
Normalization: learning rate =  0.005
Normalization: learning rate =  0.01
Normalization: learning rate =  0.02
```

[ ]:
```
plt.subplot(2, 1, 1)
plot_training_history_bn('Training accuracy (Batch Normalization)','Epoch',␣
 ↪solvers, bn_solvers, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=[' lr={:.0e}'.format(lr) for lr in lrs])
plt.subplot(2, 1, 2)
plot_training_history_bn('Validation accuracy (Batch Normalization)','Epoch',␣
 ↪solvers, bn_solvers, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=[' lr={:.0e}'.format(lr) for lr in lrs])

plt.gcf().set_size_inches(10, 15)
plt.show()
```


Training accuracy (Batch Normalization)
0.2
0.3
0.4
0.5
0.6
0
1
2
3
4
5
Epoch
w/ BN lr=5e-03
w/ BN lr=1e-02
w/ BN lr=2e-02
w/o BN lr=5e-03
w/o BN lr=1e-02
w/o BN lr=2e-02


Validation accuracy (Batch Normalization)
0.2
0.3
0.4
0.5
0
1
2
3
4
5
Epoch
w/ BN lr=5e-03
w/ BN lr=1e-02
w/ BN lr=2e-02
w/o BN lr=5e-03
w/o BN lr=1e-02
w/o BN lr=2e-02

```
[ ]:
```