FullyConnectedNets

May 24, 2023

[2]:
```
# this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = "Colab Notebooks/assignment3/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# this downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/assignment3/CV7062610/datasets
get_datasets.sh: line 5: syntax error near unexpected token `fi'
get_datasets.sh: line 5: `fi'
/content
```

[3]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from CV7062610.classifiers.fc_net import *
from CV7062610.data_utils import get_CIFAR10_data
from CV7062610.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from CV7062610.solver import Solver
```

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

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

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

```
=========== You can safely ignore the message below if you are NOT working on
ConvolutionalNetworks.ipynb ===========
        You will need to compile a Cython extension for a portion of this
assignment.
        The instructions to do this will be given in a section of the notebook
below.
        There will be an option for Colab users and another for Jupyter (local)
users.
```

[4]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

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

1 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.

2 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.

Open the file `CV7062610/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

```
[5]: from CV7062610.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

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

expected_next_w = np.asarray([
  [ 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    ]])
expected_velocity = np.asarray([
  [ 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    ]])

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

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

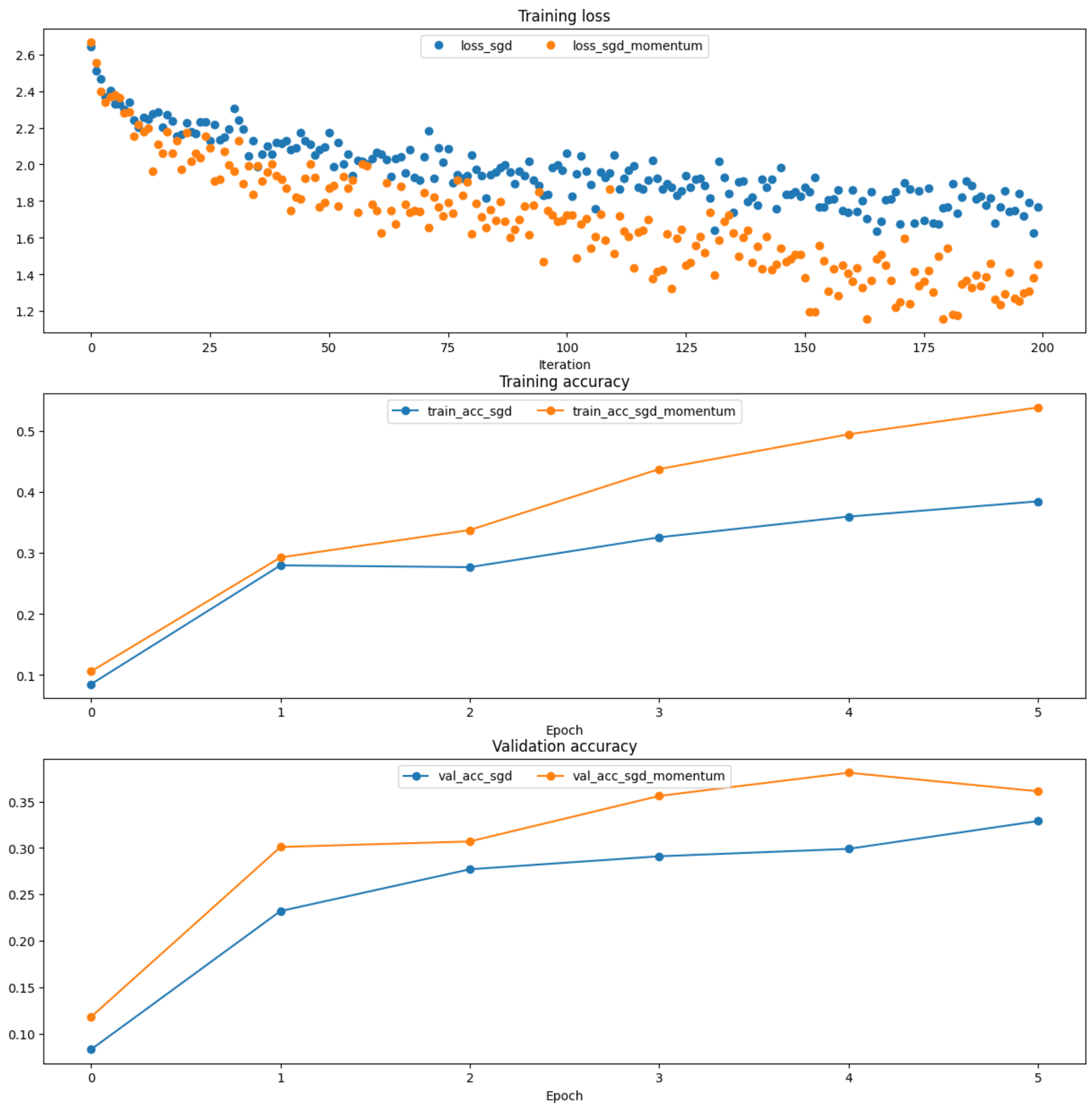
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

```
[6]: num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 5e-3,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in solvers.items():
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label="loss_%s" % update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label="train_acc_%s" % update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label="val_acc_%s" % update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.639814
(Epoch 0 / 5) train acc: 0.085000; val_acc: 0.083000
(Iteration 11 / 200) loss: 2.200271
(Iteration 21 / 200) loss: 2.228889
```

```
(Iteration 31 / 200) loss: 2.304042
(Epoch 1 / 5) train acc: 0.280000; val_acc: 0.232000
(Iteration 41 / 200) loss: 2.112324
(Iteration 51 / 200) loss: 2.173022
(Iteration 61 / 200) loss: 2.065084
(Iteration 71 / 200) loss: 2.041514
(Epoch 2 / 5) train acc: 0.277000; val_acc: 0.277000
(Iteration 81 / 200) loss: 2.051573
(Iteration 91 / 200) loss: 1.964021
(Iteration 101 / 200) loss: 2.062236
(Iteration 111 / 200) loss: 2.049165
(Epoch 3 / 5) train acc: 0.326000; val_acc: 0.291000
(Iteration 121 / 200) loss: 1.863879
(Iteration 131 / 200) loss: 1.817523
(Iteration 141 / 200) loss: 1.777860
(Iteration 151 / 200) loss: 1.877045
(Epoch 4 / 5) train acc: 0.360000; val_acc: 0.299000
(Iteration 161 / 200) loss: 1.860833
(Iteration 171 / 200) loss: 1.673111
(Iteration 181 / 200) loss: 1.766019
(Iteration 191 / 200) loss: 1.677948
(Epoch 5 / 5) train acc: 0.385000; val_acc: 0.329000

running with  sgd_momentum
(Iteration 1 / 200) loss: 2.665479
(Epoch 0 / 5) train acc: 0.106000; val_acc: 0.118000
(Iteration 11 / 200) loss: 2.215699
(Iteration 21 / 200) loss: 2.171958
(Iteration 31 / 200) loss: 1.961599
(Epoch 1 / 5) train acc: 0.293000; val_acc: 0.301000
(Iteration 41 / 200) loss: 1.917277
(Iteration 51 / 200) loss: 1.869539
(Iteration 61 / 200) loss: 1.749666
(Iteration 71 / 200) loss: 1.844760
(Epoch 2 / 5) train acc: 0.338000; val_acc: 0.307000
(Iteration 81 / 200) loss: 1.622530
(Iteration 91 / 200) loss: 1.701102
(Iteration 101 / 200) loss: 1.722402
(Iteration 111 / 200) loss: 1.516580
(Epoch 3 / 5) train acc: 0.438000; val_acc: 0.356000
(Iteration 121 / 200) loss: 1.425256
(Iteration 131 / 200) loss: 1.739389
(Iteration 141 / 200) loss: 1.553443
(Iteration 151 / 200) loss: 1.384149
(Epoch 4 / 5) train acc: 0.495000; val_acc: 0.381000
(Iteration 161 / 200) loss: 1.364395
(Iteration 171 / 200) loss: 1.248322
(Iteration 181 / 200) loss: 1.541733
```

```
(Iteration 191 / 200) loss: 1.266661
(Epoch 5 / 5) train acc: 0.539000; val_acc: 0.361000
```



3 RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `CV7062610/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

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

[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).

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

[7]:
```python
# Test RMSProp implementation
from CV7062610.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

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

expected_next_w = np.asarray([
  [-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]])
expected_cache = np.asarray([
  [ 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    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

[8]:
```python
# Test Adam implementation
from CV7062610.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

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

```
expected_next_w = np.asarray([
  [-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]])
expected_v = np.asarray([
  [ 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,   ]])
expected_m = np.asarray([
  [ 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      ]])

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

```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

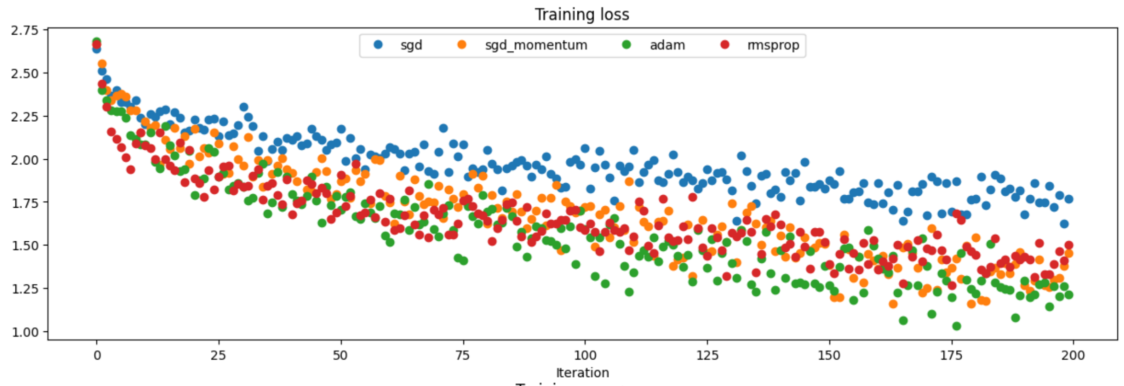
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

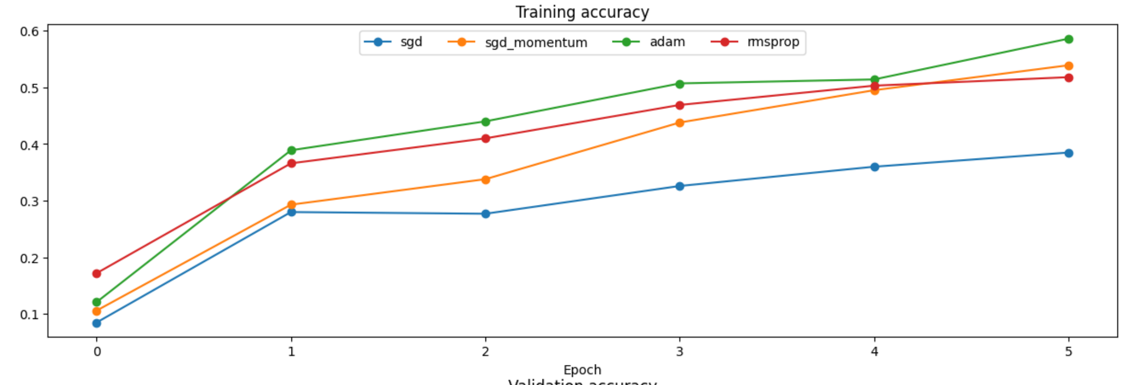
[9]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')
```

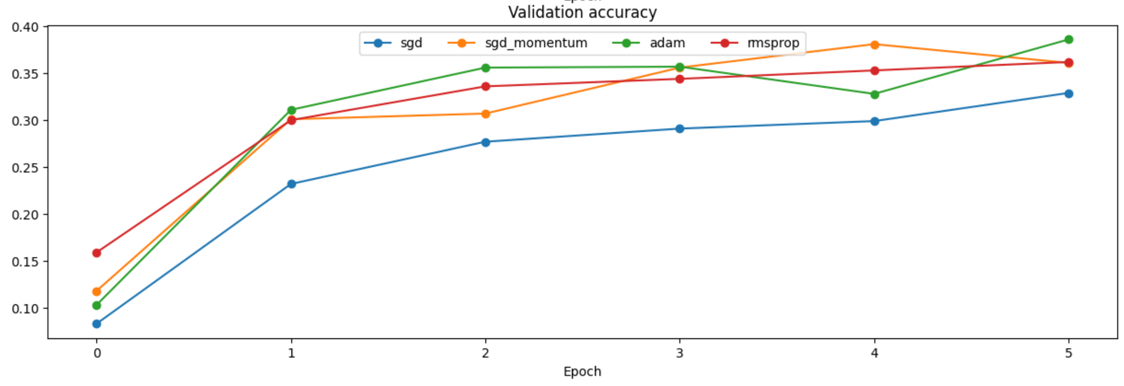
```python
plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  adam
(Iteration 1 / 200) loss: 2.681002
(Epoch 0 / 5) train acc: 0.121000; val_acc: 0.103000
(Iteration 11 / 200) loss: 2.081998
(Iteration 21 / 200) loss: 1.791508
(Iteration 31 / 200) loss: 1.836639
(Epoch 1 / 5) train acc: 0.389000; val_acc: 0.311000
(Iteration 41 / 200) loss: 1.793019
(Iteration 51 / 200) loss: 1.705007
(Iteration 61 / 200) loss: 1.517075
(Iteration 71 / 200) loss: 1.550712
(Epoch 2 / 5) train acc: 0.440000; val_acc: 0.356000
(Iteration 81 / 200) loss: 1.691616
(Iteration 91 / 200) loss: 1.609310
(Iteration 101 / 200) loss: 1.594169
(Iteration 111 / 200) loss: 1.341412
(Epoch 3 / 5) train acc: 0.507000; val_acc: 0.357000
(Iteration 121 / 200) loss: 1.374023
(Iteration 131 / 200) loss: 1.394066
(Iteration 141 / 200) loss: 1.469062
(Iteration 151 / 200) loss: 1.265040
(Epoch 4 / 5) train acc: 0.514000; val_acc: 0.328000
(Iteration 161 / 200) loss: 1.423154
```

```
(Iteration 171 / 200) loss: 1.381047
(Iteration 181 / 200) loss: 1.216678
(Iteration 191 / 200) loss: 1.294611
(Epoch 5 / 5) train acc: 0.586000; val_acc: 0.386000

running with  rmsprop
(Iteration 1 / 200) loss: 2.667155
(Epoch 0 / 5) train acc: 0.172000; val_acc: 0.159000
(Iteration 11 / 200) loss: 2.076650
(Iteration 21 / 200) loss: 1.883446
(Iteration 31 / 200) loss: 1.835276
(Epoch 1 / 5) train acc: 0.366000; val_acc: 0.300000
(Iteration 41 / 200) loss: 1.675073
(Iteration 51 / 200) loss: 1.909761
(Iteration 61 / 200) loss: 1.770841
(Iteration 71 / 200) loss: 1.675243
(Epoch 2 / 5) train acc: 0.410000; val_acc: 0.336000
(Iteration 81 / 200) loss: 1.641506
(Iteration 91 / 200) loss: 1.546858
(Iteration 101 / 200) loss: 1.586970
(Iteration 111 / 200) loss: 1.550622
(Epoch 3 / 5) train acc: 0.469000; val_acc: 0.344000
(Iteration 121 / 200) loss: 1.610663
(Iteration 131 / 200) loss: 1.572295
(Iteration 141 / 200) loss: 1.618468
(Iteration 151 / 200) loss: 1.391255
(Epoch 4 / 5) train acc: 0.503000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.357019
(Iteration 171 / 200) loss: 1.480391
(Iteration 181 / 200) loss: 1.442682
(Iteration 191 / 200) loss: 1.429294
(Epoch 5 / 5) train acc: 0.518000; val_acc: 0.362000
```







3.1 Inline Question 1:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

3.2 Answer:

The cache grows by dw^2 each iteration when using AdaGrad so the term dw/(np.sqrt(cache) + eps) converges to 0 causing the model to learn slowly. With Adam the update rule is next_w = w - lr * m_hat / (np.sqrt(v_hat) + eps) where m_hat is a bias corrected moving average of the gredient and v_hat is the bias corrected moving average of the squared gredient that way v_hat doesn't grow as fast as cache grows in comparison to m_hat and dw respectively, and thus Adam won't have the same issue.

4 Train a good model!

Train the best model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net or a cnn net.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

[10]:
```python
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

results = {}
best_val = -1

learning_rates = [0.002,0.0005]

dropout = [0.9]

hidden_dims = {"d1":[512,512,256,256]}
weight_scales = [2e-3]#,7e-2,3e-2]#

update_rules = ["sgd_momentum"]

batch_sizes = [256]
for ws in weight_scales:
    for k, dims in hidden_dims.items():
        for lr in learning_rates:
            for rule in update_rules:
                for p in dropout:
                    results[(lr, rule, k,ws,p)] = 0


                    model = FullyConnectedNet(dims,
                        weight_scale=ws, dtype=np.
↪float64,normalization="batchnorm",dropout=p)
                    solver = Solver(model, data,
                                    print_every=100, num_epochs=5,␣
↪batch_size=batch_sizes[0],
                                    update_rule=rule,
                                    optim_config={
                                    'learning_rate': lr,
                                    }
                            )
                    solver.train()

                    val_accuracy = solver.best_val_acc

                    results[(lr, rule, k,ws,p)] = val_accuracy
                    if(val_accuracy > best_val):
                        best_val = val_accuracy
                        best_model = model

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 955) loss: 2.302257
(Epoch 0 / 5) train acc: 0.092000; val_acc: 0.108000
(Iteration 101 / 955) loss: 1.576851
(Epoch 1 / 5) train acc: 0.483000; val_acc: 0.473000
(Iteration 201 / 955) loss: 1.452330
(Iteration 301 / 955) loss: 1.391654
(Epoch 2 / 5) train acc: 0.524000; val_acc: 0.499000
(Iteration 401 / 955) loss: 1.323220
(Iteration 501 / 955) loss: 1.195332
(Epoch 3 / 5) train acc: 0.576000; val_acc: 0.531000
(Iteration 601 / 955) loss: 1.340125
(Iteration 701 / 955) loss: 1.184022
(Epoch 4 / 5) train acc: 0.581000; val_acc: 0.524000
(Iteration 801 / 955) loss: 1.206341
(Iteration 901 / 955) loss: 1.248851
(Epoch 5 / 5) train acc: 0.650000; val_acc: 0.537000
(Iteration 1 / 955) loss: 2.304473
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.102000
(Iteration 101 / 955) loss: 1.881269
(Epoch 1 / 5) train acc: 0.430000; val_acc: 0.442000
(Iteration 201 / 955) loss: 1.562678
(Iteration 301 / 955) loss: 1.536673
(Epoch 2 / 5) train acc: 0.513000; val_acc: 0.488000
```

```
(Iteration 401 / 955) loss: 1.429823
(Iteration 501 / 955) loss: 1.365188
(Epoch 3 / 5) train acc: 0.579000; val_acc: 0.523000
(Iteration 601 / 955) loss: 1.303100
(Iteration 701 / 955) loss: 1.120499
(Epoch 4 / 5) train acc: 0.600000; val_acc: 0.538000
(Iteration 801 / 955) loss: 1.261536
(Iteration 901 / 955) loss: 1.087911
(Epoch 5 / 5) train acc: 0.614000; val_acc: 0.537000
```

5 Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

[11]:
```python
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.525
Test set accuracy:  0.513
```

[11]:
```python

```

BatchNormalization

May 24, 2023

[1]:
```
# this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = "Colab Notebooks/assignment3/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# this downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/assignment3/CV7062610/datasets
get_datasets.sh: line 6: syntax error: unexpected end of file
/content
```

1 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.

[2]:
```
# As usual, a bit of setup
import time
import numpy as np
import matplotlib.pyplot as plt
from CV7062610.classifiers.fc_net import *
from CV7062610.data_utils import get_CIFAR10_data
from CV7062610.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from CV7062610.solver import Solver

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

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

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

def print_mean_std(x,axis=0):
    print('  means: ', x.mean(axis=axis))
    print('  stds:  ', x.std(axis=axis))
    print()
```

```
=========== You can safely ignore the message below if you are NOT working on
ConvolutionalNetworks.ipynb ===========
        You will need to compile a Cython extension for a portion of this
assignment.
        The instructions to do this will be given in a section of the notebook
below.
        There will be an option for Colab users and another for Jupyter (local)
users.
```

[3]:
```python
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

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

1.1 Batch normalization: forward

In the file `CV7062610/layers.py`, 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!

[4]:
```python
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

# Simulate the forward pass for a two-layer network
np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

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

gamma = np.ones((D3,))
beta = np.zeros((D3,))
# Means should be close to zero and stds close to one
print('After batch normalization (gamma=1, beta=0)')
```

```
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
# Now means should be close to beta and stds close to gamma
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)
```

```
Before batch normalization:
  means:  [ -2.3814598  -13.18038246   1.91780462]
  stds:   [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)
  means:  [ 3.55271368e-17  1.71529457e-16 -2.76167977e-17]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means:  [11. 12. 13.]
  stds:   [0.99999999 1.99999999 2.99999999]
```

[5]:
```
# 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.

np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)

bn_param = {'mode': 'train'}
gamma = np.ones(D3)
beta = np.zeros(D3)

for t in range(50):
  X = np.random.randn(N, D1)
  a = np.maximum(0, X.dot(W1)).dot(W2)
  batchnorm_forward(a, gamma, beta, bn_param)

bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(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,axis=0)
```

```
After batch normalization (test-time):
  means:   [-0.03927354 -0.04349152 -0.10452688]
  stds:    [1.01531428 1.01238373 0.97819988]
```

1.2 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.

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

[6]:
```
# Gradient check batchnorm backward pass
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

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 = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
db_num = eval_numerical_gradient_array(fb, beta.copy(), 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-13 and 1e-8
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.7029218575884033e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

1.3 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 $X = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_N \end{bmatrix}$,

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.

[7]:
```
np.random.seed(231)
N, D = 100, 500
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
```

```
dout = np.random.randn(N, D)

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

```
dx difference:  1.209751841653115e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.63x
```

1.4 Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `CV7062610/classifiers/fc_net.py`. Modify your implementation to add batch normalization.

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

HINT: You might find it useful to define an additional helper layer similar to those in the file `CV7062610/layer_utils.py`. If you decide to do so, do it in the file `CV7062610/classifiers/fc_net.py`.

[8]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

# You should expect losses between 1e-4~1e-10 for W,
# losses between 1e-08~1e-10 for b,
# and losses between 1e-08~1e-09 for beta and gammas.
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=np.float64,
                            normalization='batchnorm')
  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
↪h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 2.85e-06
W3 relative error: 4.14e-10
b1 relative error: 3.11e-07
b2 relative error: 1.94e-08
b3 relative error: 8.72e-11
beta1 relative error: 6.94e-09
beta2 relative error: 1.17e-09
gamma1 relative error: 7.47e-09
gamma2 relative error: 1.56e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 1.11e-08
b1 relative error: 1.38e-08
b2 relative error: 7.99e-07
b3 relative error: 1.42e-10
beta1 relative error: 6.65e-09
beta2 relative error: 3.48e-09
gamma1 relative error: 6.27e-09
gamma2 relative error: 5.28e-09
```

2 Batchnorm for deep networks

Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

[9]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000
```

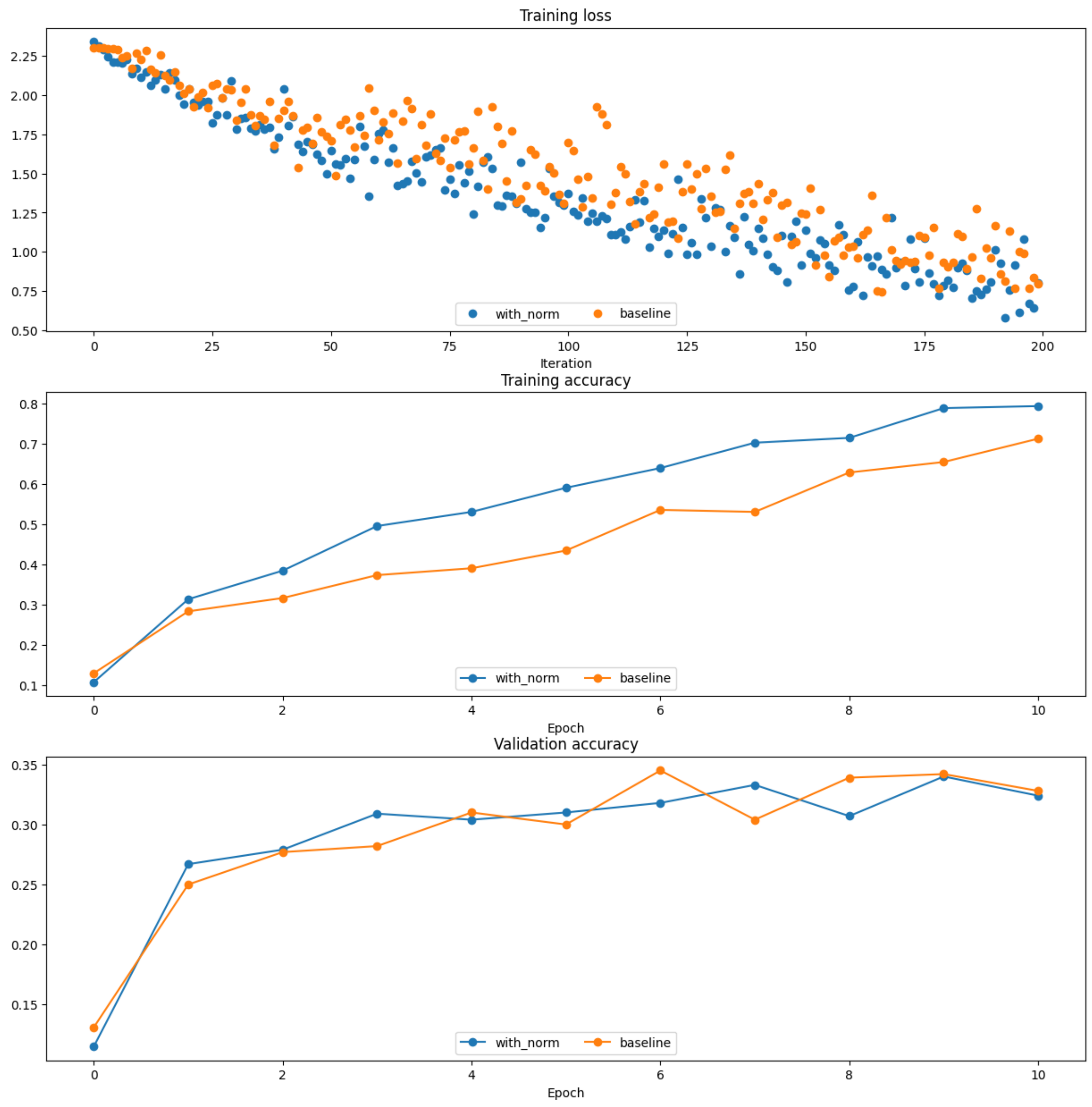
```
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 2e-2
bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization='batchnorm')
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None)

print('Solver with batch norm:')
bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True,print_every=20)
bn_solver.train()

print('\nSolver without batch norm:')
solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.313000; val_acc: 0.267000
(Iteration 21 / 200) loss: 2.039365
(Epoch 2 / 10) train acc: 0.384000; val_acc: 0.279000
(Iteration 41 / 200) loss: 2.041103
(Epoch 3 / 10) train acc: 0.495000; val_acc: 0.309000
(Iteration 61 / 200) loss: 1.753902
(Epoch 4 / 10) train acc: 0.530000; val_acc: 0.304000
(Iteration 81 / 200) loss: 1.241217
(Epoch 5 / 10) train acc: 0.590000; val_acc: 0.310000
(Iteration 101 / 200) loss: 1.371013
(Epoch 6 / 10) train acc: 0.639000; val_acc: 0.318000
```

```
(Iteration 121 / 200) loss: 1.135679
(Epoch 7 / 10) train acc: 0.702000; val_acc: 0.333000
(Iteration 141 / 200) loss: 1.152047
(Epoch 8 / 10) train acc: 0.714000; val_acc: 0.307000
(Iteration 161 / 200) loss: 0.781041
(Epoch 9 / 10) train acc: 0.788000; val_acc: 0.340000
(Iteration 181 / 200) loss: 0.820985
(Epoch 10 / 10) train acc: 0.793000; val_acc: 0.324000

Solver without batch norm:
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.131000
(Epoch 1 / 10) train acc: 0.283000; val_acc: 0.250000
(Iteration 21 / 200) loss: 2.041970
(Epoch 2 / 10) train acc: 0.316000; val_acc: 0.277000
(Iteration 41 / 200) loss: 1.900473
(Epoch 3 / 10) train acc: 0.373000; val_acc: 0.282000
(Iteration 61 / 200) loss: 1.713156
(Epoch 4 / 10) train acc: 0.390000; val_acc: 0.310000
(Iteration 81 / 200) loss: 1.662210
(Epoch 5 / 10) train acc: 0.434000; val_acc: 0.300000
(Iteration 101 / 200) loss: 1.696058
(Epoch 6 / 10) train acc: 0.535000; val_acc: 0.345000
(Iteration 121 / 200) loss: 1.557986
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.304000
(Iteration 141 / 200) loss: 1.432189
(Epoch 8 / 10) train acc: 0.628000; val_acc: 0.339000
(Iteration 161 / 200) loss: 1.034116
(Epoch 9 / 10) train acc: 0.654000; val_acc: 0.342000
(Iteration 181 / 200) loss: 0.905796
(Epoch 10 / 10) train acc: 0.712000; val_acc: 0.328000
```

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.

[10]:
```python
def plot_training_history(title, label, baseline, 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_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


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

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



3 Batch normalization and initialization

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

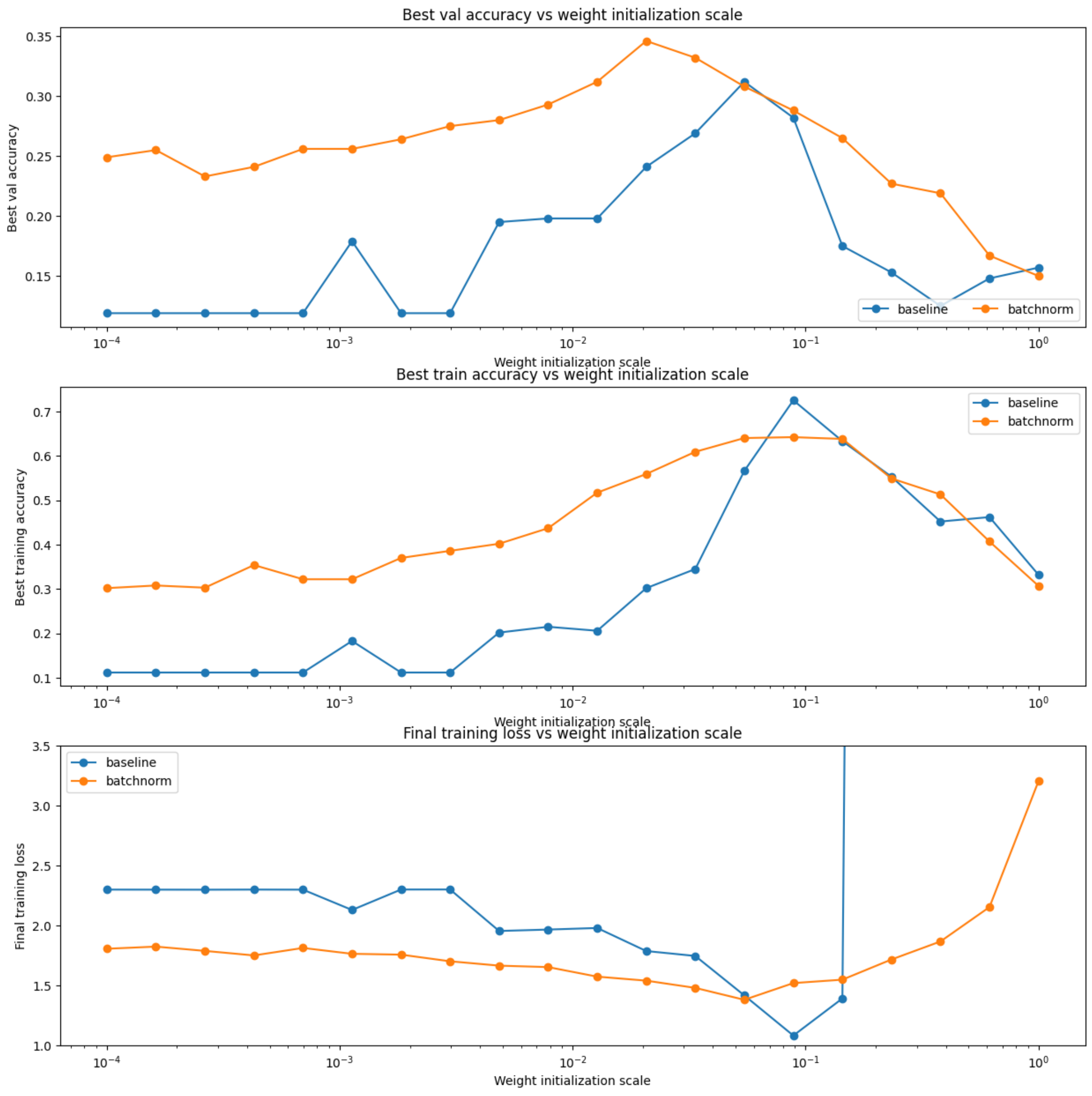
The first cell will train 8-layer networks both with and without batch normalization using different scales for weight initialization. The second layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

[11]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
```

```
num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

bn_solvers_ws = {}
solvers_ws = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
    print('Running weight scale %d / %d' % (i + 1, len(weight_scales)))
    bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization='batchnorm')
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None)

    bn_solver = Solver(bn_model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 1e-3,
                  },
                  verbose=False, print_every=200)
    bn_solver.train()
    bn_solvers_ws[weight_scale] = bn_solver

    solver = Solver(model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 1e-3,
                  },
                  verbose=False, print_every=200)
    solver.train()
    solvers_ws[weight_scale] = solver
```

```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

[12]:
```
# Plot results of weight scale experiment
best_train_accs, bn_best_train_accs = [], []
best_val_accs, bn_best_val_accs = [], []
final_train_loss, bn_final_train_loss = [], []

for ws in weight_scales:
  best_train_accs.append(max(solvers_ws[ws].train_acc_history))
  bn_best_train_accs.append(max(bn_solvers_ws[ws].train_acc_history))

  best_val_accs.append(max(solvers_ws[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers_ws[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers_ws[ws].loss_history[-100:]))
  bn_final_train_loss.append(np.mean(bn_solvers_ws[ws].loss_history[-100:]))

plt.subplot(3, 1, 1)
plt.title('Best val accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best val accuracy')
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
plt.title('Best train accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best training accuracy')
plt.semilogx(weight_scales, best_train_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_train_accs, '-o', label='batchnorm')
plt.legend()

plt.subplot(3, 1, 3)
plt.title('Final training loss vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()
plt.gca().set_ylim(1.0, 3.5)

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



3.1 Inline Question 1:

Describe the results of this experiment. How does the scale of weight initialization affect models with/without batch normalization differently, and why?

3.2 Answer:

Adding batch normalization makes each layer input much less noisy thus making outliers have less of an effect on the network, this allows us to use larger scales since the far outliers don't get multiplied by very large weights

4 Batch normalization and batch size

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

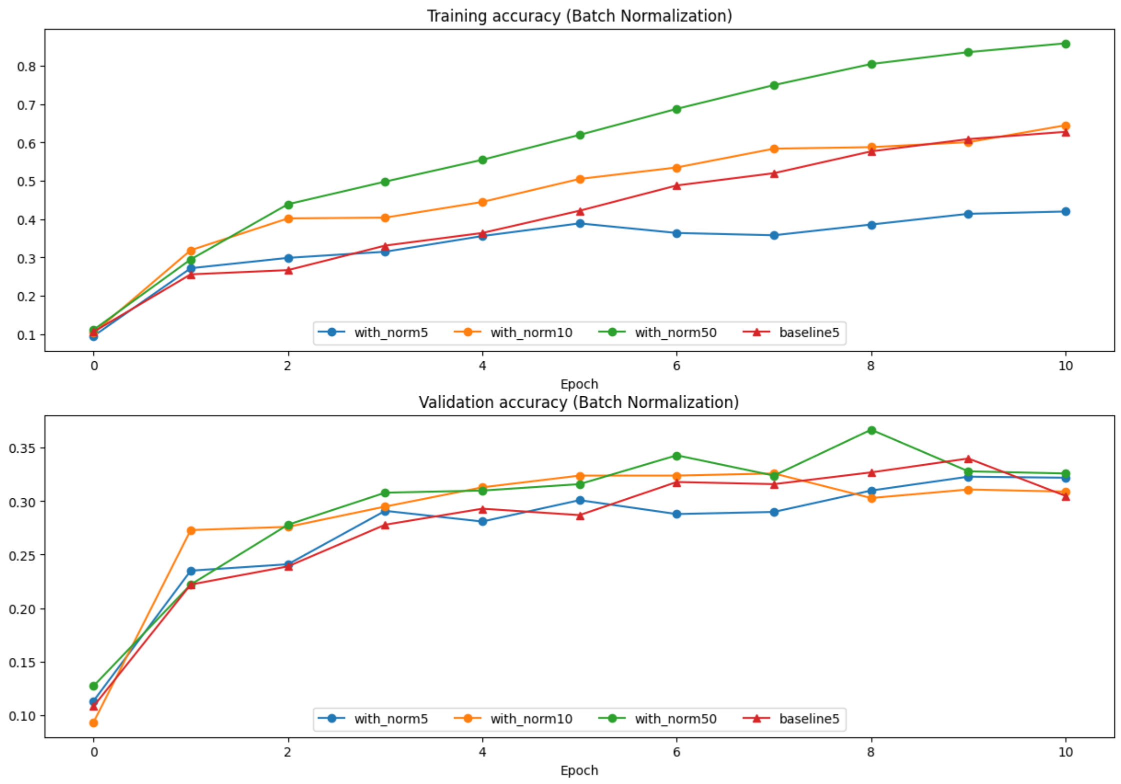
The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

[13]:
```
def run_batchsize_experiments(normalization_mode):
    np.random.seed(231)
    # Try training a very deep net with batchnorm
    hidden_dims = [100, 100, 100, 100, 100]
    num_train = 1000
    small_data = {
      'X_train': data['X_train'][:num_train],
      'y_train': data['y_train'][:num_train],
      'X_val': data['X_val'],
      'y_val': data['y_val'],
    }
    n_epochs=10
    weight_scale = 2e-2
    batch_sizes = [5,10,50]
    lr = 10**(-3.5)
    solver_bsize = batch_sizes[0]

    print('No normalization: batch size = ',solver_bsize)
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
  ↪normalization=None)
    solver = Solver(model, small_data,
                    num_epochs=n_epochs, batch_size=solver_bsize,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': lr,
                    },
                    verbose=False)
    solver.train()

    bn_solvers = []
    for i in range(len(batch_sizes)):
        b_size=batch_sizes[i]
        print('Normalization: batch size = ',b_size)
        bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
  ↪normalization=normalization_mode)
```

```
        bn_solver = Solver(bn_model, small_data,
                        num_epochs=n_epochs, batch_size=b_size,
                        update_rule='adam',
                        optim_config={
                          'learning_rate': lr,
                        },
                        verbose=False)
        bn_solver.train()
        bn_solvers.append(bn_solver)

    return bn_solvers, solver, batch_sizes

batch_sizes = [5,10,50]
bn_solvers_bsize, solver_bsize, batch_sizes =␣
 ↪run_batchsize_experiments('batchnorm')
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```

[14]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)

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



4.1 Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

4.2 Answer:

The larger the batch the more accurate our model is, since we scale the data by the running mean and running variance of each batch. Larger batches result in more accurate mean and variance.

5 Layer Normalization

Batch normalization has proved to be effective in making networks easier to train, but the dependency on batch size makes it less useful in complex networks which have a cap on the input batch size due to hardware limitations.

Several alternatives to batch normalization have been proposed to mitigate this problem; one such technique is Layer Normalization [2]. Instead of normalizing over the batch, we normalize over the features. In other words, when using Layer Normalization, each feature vector corresponding to a single datapoint is normalized based on the sum of all terms within that feature vector.

[2] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.

5.1 Inline Question 3:

Which of these data preprocessing steps is analogous to batch normalization, and which is analogous to layer normalization?

1. Scaling each image in the dataset, so that the RGB channels for each row of pixels within an image sums up to 1.
2. Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1.

3. Subtracting the mean image of the dataset from each image in the dataset.
4. Setting all RGB values to either 0 or 1 depending on a given threshold.

5.2 Answer:

2 is analogous to layer normalization 3 is analogous to layer batch normalization

6 Layer Normalization: Implementation

Now you'll implement layer normalization. This step should be relatively straightforward, as conceptually the implementation is almost identical to that of batch normalization. One significant difference though is that for layer normalization, we do not keep track of the moving moments, and the testing phase is identical to the training phase, where the mean and variance are directly calculated per datapoint.

Here's what you need to do:

- In `CV7062610/layers.py`, implement the forward pass for layer normalization in the function `layernorm_forward`.

Run the cell below to check your results. * In `CV7062610/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `CV7062610/classifiers/fc_net.py` to add layer normalization to the `FullyConnectedNet`. When the `normalization` flag is set to `"layernorm"` in the constructor, you should insert a layer normalization layer before each ReLU nonlinearity.

Run the third cell below to run the batch size experiment on layer normalization.

[15]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after layer normalization

# Simulate the forward pass for a two-layer network
np.random.seed(231)
N, D1, D2, D3 =4, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)
```

```
print('Before layer normalization:')
print_mean_std(a,axis=1)

gamma = np.ones(D3)
beta = np.zeros(D3)
# Means should be close to zero and stds close to one
print('After layer normalization (gamma=1, beta=0)')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)

gamma = np.asarray([3.0,3.0,3.0])
beta = np.asarray([5.0,5.0,5.0])
# Now means should be close to beta and stds close to gamma
print('After layer normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)
```

```
Before layer normalization:
  means:  [-59.06673243 -47.60782686 -43.31137368 -26.40991744]
  stds:   [10.07429373 28.39478981 35.28360729  4.01831507]

After layer normalization (gamma=1, beta=0)
  means:  [ 2.59052039e-16  0.00000000e+00  2.22044605e-16 -5.55111512e-16]
  stds:   [0.99999995 0.99999999 1.         0.99999969]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
  means:  [5. 5. 5. 5.]
  stds:   [2.99999985 2.99999998 2.99999999 2.99999907]
```

[16]:
```
# Gradient check batchnorm backward pass
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

ln_param = {}
fx = lambda x: layernorm_forward(x, gamma, beta, ln_param)[0]
fg = lambda a: layernorm_forward(x, a, beta, ln_param)[0]
fb = lambda b: layernorm_forward(x, gamma, b, ln_param)[0]

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

```
_, cache = layernorm_forward(x, gamma, beta, ln_param)
dx, dgamma, dbeta = layernorm_backward(dout, cache)

#You should expect to see relative errors between 1e-12 and 1e-8
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

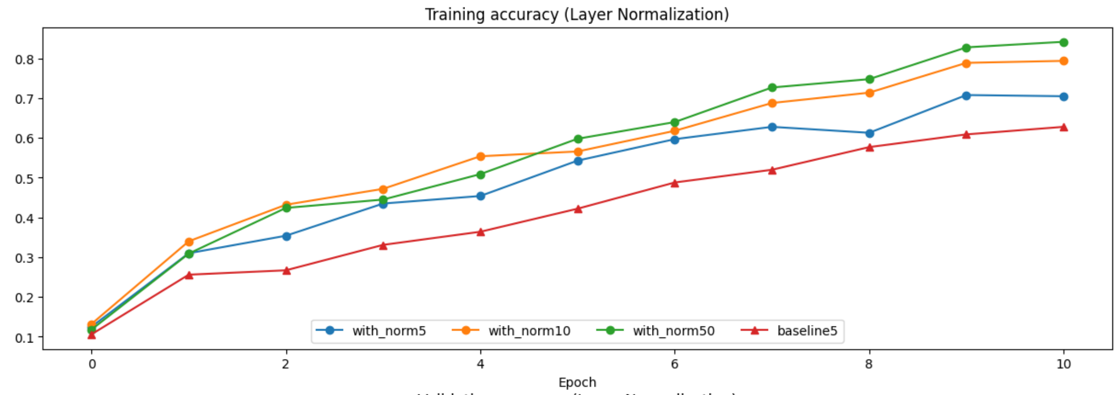
```
dx error:  1.4336160411201157e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12
```

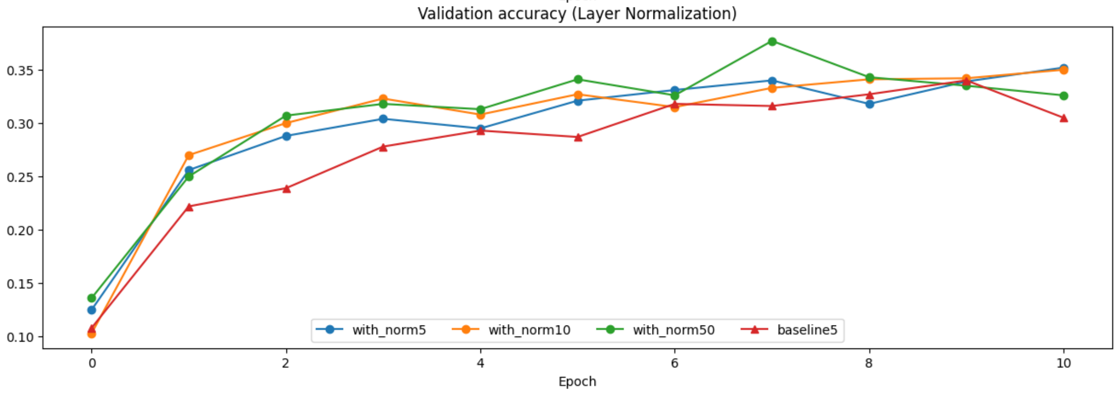
7 Layer Normalization and batch size

We will now run the previous batch size experiment with layer normalization instead of batch normalization. Compared to the previous experiment, you should see a markedly smaller influence of batch size on the training history!

[17]:
```
ln_solvers_bsize, solver_bsize, batch_sizes =␣
  ↪run_batchsize_experiments('layernorm')

plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Layer Normalization)','Epoch',␣
  ↪solver_bsize, ln_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
  ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Layer Normalization)','Epoch',␣
  ↪solver_bsize, ln_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
  ↪bn_marker='-o', labels=batch_sizes)

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

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```





7.1 Inline Question 4:

When is layer normalization likely to not work well, and why?

1. Using it in a very deep network
2. Having a very small dimension of features
3. Having a high regularization term

7.2 Answer:

1. since we lose some accuracy during layer normalization, we will continue to accumulate the errors with each layer, thus causing our network to "blur" out our input data too much
2. with a very small amount of dimensions the mean and variance may be vary far from the actual values of the features, this could cause layer normalization not to work correctly
3. since also layer normalization also acts as a regularizer, it might cause the network to be overly restricted leading to underfitting

8 Spatial batch normalization: forward

In the file CV7062610/layers.py, implement the forward pass for spatial batch normalization in the function spatial_batchnorm_forward. Check your implementation by running the following:

[18]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

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

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

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

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

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [9.33463814 8.90909116 9.11056338]
  Stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 6.18949336e-16  5.99520433e-16 -1.22124533e-16]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

[19]:
```
np.random.seed(231)
# 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 = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_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(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

9 Spatial batch normalization: backward

In the file CV7062610/layers.py, implement the backward pass for spatial batch normalization in the function spatial_batchnorm_backward. Run the following to check your implementation using a numeric gradient check:

[20]:
```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

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

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

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
```

```
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  2.7866481899887757e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

Dropout

May 24, 2023

[1]:
```
# this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = "Colab Notebooks/assignment3/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# this downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/assignment3/CV7062610/datasets
get_datasets.sh: line 5: syntax error near unexpected token `fi'
get_datasets.sh: line 5: `fi'
/content
```

1 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

[2]:
```
# As usual, a bit of setup
from __future__ import print_function
```

```
import time
import numpy as np
import matplotlib.pyplot as plt
from CV7062610.classifiers.fc_net import *
from CV7062610.data_utils import get_CIFAR10_data
from CV7062610.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from CV7062610.solver import Solver

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

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

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

```
=========== You can safely ignore the message below if you are NOT working on
ConvolutionalNetworks.ipynb ===========
        You will need to compile a Cython extension for a portion of this
assignment.
        The instructions to do this will be given in a section of the notebook
below.
        There will be an option for Colab users and another for Jupyter (local)
users.
```

[3]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

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

[3]:
```

```

2 Dropout forward pass

In the file `CV7062610/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

[4]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 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())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

3 Dropout backward pass

In the file `CV7062610/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

```
[5]: np.random.seed(231)
     x = np.random.randn(10, 10) + 10
     dout = np.random.randn(*x.shape)

     dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
     out, cache = dropout_forward(x, dropout_param)
     dx = dropout_backward(dout, cache)
     dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,␣
      ↪dropout_param)[0], x, dout)

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

```
dx relative error:  5.44560814873387e-11
```

3.1 Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

3.2 Answer:

during training each neuron's expected output would be `px + (1-p)*0`, and not `x` since we dropped `p` neurons, so when we now test the model, the expected output would change to `x` instead of `px` and the inputing neurons won't be able to handle the change. by dividing by `p` during training our expected output would be `x` and we won't have to adjust `x` in the testing phase.

4 Fully-connected nets with Dropout

In the file `CV7062610/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the network receives a value that is not 1 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.

```
[6]: np.random.seed(231)
     N, D, H1, H2, C = 2, 15, 20, 30, 10
     X = np.random.randn(N, D)
     y = np.random.randint(C, size=(N,))

     for dropout in [1, 0.75, 0.5]:
       print('Running check with dropout = ', dropout)
       model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                                 weight_scale=5e-2, dtype=np.float64,
                                 dropout=dropout, seed=123)

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

```
  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
↪h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11

Running check with dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.90e-07
W2 relative error: 4.76e-06
W3 relative error: 2.60e-08
b1 relative error: 4.73e-09
b2 relative error: 1.82e-09
b3 relative error: 1.70e-10

Running check with dropout =  0.5
Initial loss:  2.3042759220785896
W1 relative error: 3.11e-07
W2 relative error: 1.84e-08
W3 relative error: 5.35e-08
b1 relative error: 5.37e-09
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

5 Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

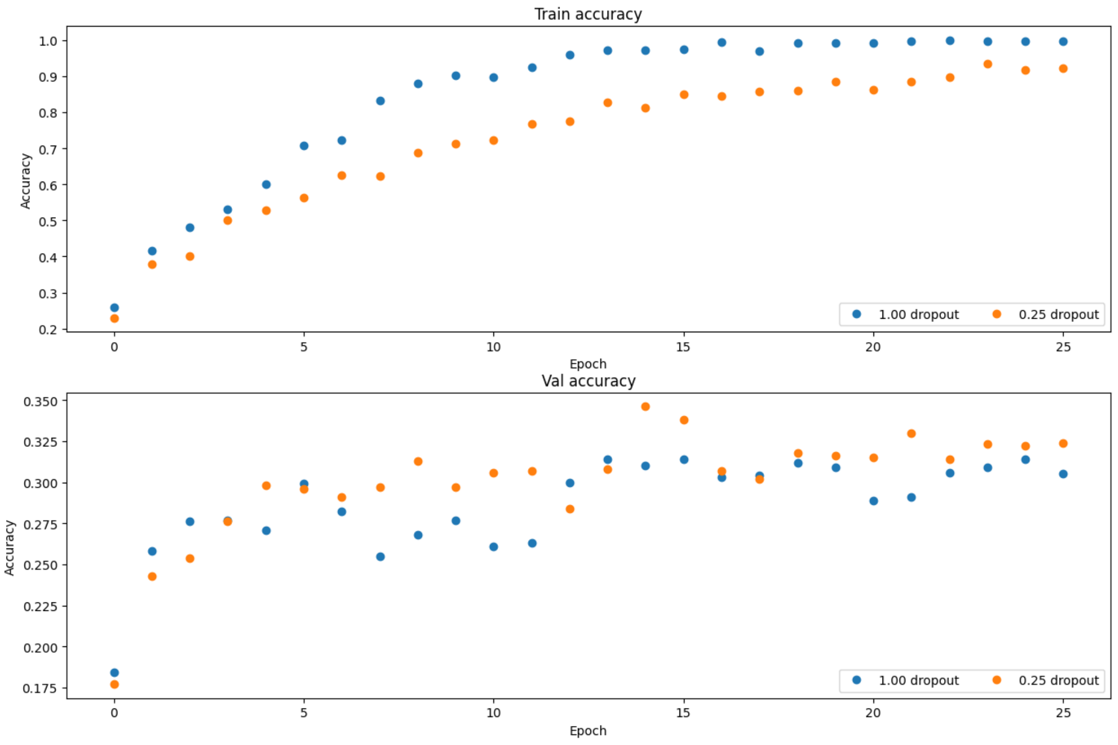
[7]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
  print()
```

```
1
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.880000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.277000
(Epoch 10 / 25) train acc: 0.898000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.924000; val_acc: 0.263000
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.972000; val_acc: 0.314000
(Epoch 14 / 25) train acc: 0.972000; val_acc: 0.310000
(Epoch 15 / 25) train acc: 0.974000; val_acc: 0.314000
(Epoch 16 / 25) train acc: 0.994000; val_acc: 0.303000
(Epoch 17 / 25) train acc: 0.970000; val_acc: 0.304000
(Epoch 18 / 25) train acc: 0.992000; val_acc: 0.312000
(Epoch 19 / 25) train acc: 0.992000; val_acc: 0.309000
(Epoch 20 / 25) train acc: 0.992000; val_acc: 0.289000
(Iteration 101 / 125) loss: 0.001969
```

```
(Epoch 21 / 25) train acc: 0.996000; val_acc: 0.291000
(Epoch 22 / 25) train acc: 1.000000; val_acc: 0.306000
(Epoch 23 / 25) train acc: 0.996000; val_acc: 0.309000
(Epoch 24 / 25) train acc: 0.998000; val_acc: 0.314000
(Epoch 25 / 25) train acc: 0.998000; val_acc: 0.305000

0.25
(Iteration 1 / 125) loss: 17.318478
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.296000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.297000
(Epoch 8 / 25) train acc: 0.688000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.712000; val_acc: 0.297000
(Epoch 10 / 25) train acc: 0.724000; val_acc: 0.306000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.307000
(Epoch 12 / 25) train acc: 0.774000; val_acc: 0.284000
(Epoch 13 / 25) train acc: 0.828000; val_acc: 0.308000
(Epoch 14 / 25) train acc: 0.812000; val_acc: 0.346000
(Epoch 15 / 25) train acc: 0.850000; val_acc: 0.338000
(Epoch 16 / 25) train acc: 0.844000; val_acc: 0.307000
(Epoch 17 / 25) train acc: 0.858000; val_acc: 0.302000
(Epoch 18 / 25) train acc: 0.860000; val_acc: 0.318000
(Epoch 19 / 25) train acc: 0.884000; val_acc: 0.316000
(Epoch 20 / 25) train acc: 0.862000; val_acc: 0.315000
(Iteration 101 / 125) loss: 4.293572
(Epoch 21 / 25) train acc: 0.886000; val_acc: 0.330000
(Epoch 22 / 25) train acc: 0.898000; val_acc: 0.314000
(Epoch 23 / 25) train acc: 0.934000; val_acc: 0.323000
(Epoch 24 / 25) train acc: 0.918000; val_acc: 0.322000
(Epoch 25 / 25) train acc: 0.922000; val_acc: 0.324000
```

[8]:
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
```

```
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' %␣
  ↪dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' %␣
  ↪dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

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



5.1 Inline Question 2:

Compare the validation and training accuracies with and without dropout – what do your results suggest about dropout as a regularizer?

5.2 Answer:

we can see that although the training accuracy with dropout is higher, the validation accuracy is lower suggesting that dropout managed to stop the model from overfitting too much, this shows that dropout may work as a regularizer

5.3 Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). If we are concerned about overfitting, how should we modify p (if at all) when we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

5.4 Answer:

since decreasing the size of hidden layers should help prevent overfitting, we shouldn't change the value of `p` unless the model is underfitting or overfitting. note that the new value of `p` should be

`new_p < old_p*n_layers/n_newlayers`

[8]:
```

```

PyTorch

May 24, 2023

[27]:
```
# this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = "Colab Notebooks/assignment3/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# this downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/assignment3/CV7062610/datasets
get_datasets.sh: line 6: syntax error: unexpected end of file
/content
```

1 What's this PyTorch business?

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you choose to use that notebook).

1.0.1 What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

1.0.2 Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.0.3 PyTorch versions

This notebook assumes that you are using **PyTorch version 1.4**. In some of the previous versions (e.g. before 0.4), Tensors had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 1.0+ versions separate a Tensor's datatype from its device, and use numpy-style factories for constructing Tensors rather than directly invoking Tensor constructors.

1.1 How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

1.2 Install PyTorch 1.4 (ONLY IF YOU ARE WORKING LOCALLY)

1. Have the latest version of Anaconda installed on your machine.
2. Create a new conda environment starting from Python 3.7. In this setup example, we'll call it `torch_env`.
3. Run the command: `conda activate torch_env`
4. Run the command: `pip install torch==1.4 torchvision==0.5.0`

2 Table of Contents

This assignment has 5 parts. You will learn PyTorch on **three different levels of abstraction**, which will help you understand it better and prepare you for the final project.

1. Part I, Preparation: we will use CIFAR-10 dataset.

2. Part II, Barebones PyTorch: **Abstraction level 1**, we will work directly with the lowest-level PyTorch Tensors.
3. Part III, PyTorch Module API: **Abstraction level 2**, we will use `nn.Module` to define arbitrary neural network architecture.
4. Part IV, PyTorch Sequential API: **Abstraction level 3**, we will use `nn.Sequential` to define a linear feed-forward network very conveniently.
5. Part V, CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`nn.Module`	High	Medium
`nn.Sequential`	Low	High

3 Part I. Preparation

First, we load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

[28]:
```
import torch
#assert '.'.join(torch.__version__.split('.')[:2]) == '1.4'
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np
```

[29]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
```

```
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./CV7062610/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./CV7062610/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN,␣
 ↪50000)))

cifar10_test = dset.CIFAR10('./CV7062610/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

You have an option to **use GPU by setting the flag to True below**. It is not necessary to use GPU for this assignment. Note that if your computer does not have CUDA enabled, `torch.cuda.is_available()` will return False and this notebook will fallback to CPU mode.

The global variables `dtype` and `device` will control the data types throughout this assignment.

3.1 Colab Users

If you are using Colab, you need to manually switch to a GPU device. You can do this by clicking `Runtime -> Change runtime type` and selecting `GPU` under `Hardware Accelerator`. Note that you have to rerun the cells from the top since the kernel gets restarted upon switching runtimes.

[30]:
```
USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100
```

```
print('using device:', device)
```

```
using device: cuda
```

4 Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the scalar loss at the end.

4.0.1 PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector – it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

[31]:
```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector␣
↪per image

def test_flatten():
    x = torch.arange(12).view(2, 1, 3, 2)
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[ 0,  1],
          [ 2,  3],
          [ 4,  5]]],


        [[[ 6,  7],
          [ 8,  9],
          [10, 11]]]])
After flattening:  tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
```

4.0.2 Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

[32]:
```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H␣
↪units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the network;
      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification scores for
      the input data x.
    """
    # first we flatten the image
    x = flatten(x)  # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Since w1␣
  ↪and
    # w2 have requires_grad=True, operations involving these Tensors will cause
    # PyTorch to build a computational graph, allowing automatic computation of
    # gradients. Since we are no longer implementing the backward pass by hand␣
  ↪we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
    x = F.relu(x.mm(w1))
    x = x.mm(w2)
    return x


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature␣
  ↪dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

4.0.3 Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity

3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

Note that we have **no softmax activation** here after our fully-connected layer: this is because PyTorch's cross entropy loss performs a softmax activation for you, and by bundling that step in makes computation more efficient.

HINT: For convolutions: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; pay attention to the shapes of convolutional filters!

[33]:
```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the␣
↪first
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2) giving
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the␣
↪second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can␣
↪you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can␣
↪you
        figure out what the shape should be?

    Returns:
    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
   ␣
↪################################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.            ␣
↪   #
```

```
  ␣
↪################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    scores = F.relu(F.conv2d(x,conv_w1,bias=conv_b1,stride=1,padding=2))
    scores = F.relu(F.conv2d(scores,conv_w2,bias=conv_b2,stride=1,padding=1))
    scores = flatten(scores)
    scores = scores.mm(fc_w).add(fc_b)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  ␣
↪################################################################################
    #                                END OF YOUR CODE                              ␣
↪   #
  ␣
↪################################################################################
    return scores
```

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

When you run this function, scores should have shape (64, 10).

[34]:
```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
↪size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel,␣
↪in_channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel,␣
↪in_channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before␣
↪the fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w,␣
↪fc_b])
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

4.0.4 Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The `random_weight` function uses the Kaiming normal initialization method, described in:

He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015, https://arxiv.org/abs/1502.01852

[35]:
```python
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH,␣
 ↪kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

[35]:
```
tensor([[ 0.2425,  0.2477, -0.0163, -0.3513,  0.4410],
        [-0.9614, -0.5442, -1.4061,  0.6368, -1.0309],
        [-0.1936, -0.0195, -1.0113, -1.5479,  0.5831]], device='cuda:0',
       requires_grad=True)
```

4.0.5 Barebones PyTorch: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

[36]:
```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *␣
 ↪acc))
```

4.0.6 BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can read about it here.

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

[37]:
```
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model.
      It should have the signature scores = model_fn(x, params) where x is a
      PyTorch Tensor of image data, params is a list of PyTorch Tensors giving
      model weights, and scores is a PyTorch Tensor of shape (N, C) giving
      scores for the elements in x.
    - params: List of PyTorch Tensors giving weights for the model
    - learning_rate: Python scalar giving the learning rate to use for SGD
    Returns: Nothing
    """
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss
        scores = model_fn(x, params)
        loss = F.cross_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational
        # graph has requires_grad=True and uses backpropagation to compute the
        # gradient of the loss with respect to these Tensors, and stores the
        # gradients in the .grad attribute of each Tensor.
        loss.backward()

        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the backward pass
                w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

4.0.7 BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 40% after training for one epoch.

[38]:
```python
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 3.1108
Checking accuracy on the val set
Got 167 / 1000 correct (16.70%)

Iteration 100, loss = 2.1952
Checking accuracy on the val set
Got 343 / 1000 correct (34.30%)

Iteration 200, loss = 2.2809
Checking accuracy on the val set
Got 357 / 1000 correct (35.70%)

Iteration 300, loss = 1.7922
Checking accuracy on the val set
Got 391 / 1000 correct (39.10%)

Iteration 400, loss = 1.4591
Checking accuracy on the val set
Got 426 / 1000 correct (42.60%)

Iteration 500, loss = 1.5613
Checking accuracy on the val set
Got 427 / 1000 correct (42.70%)

Iteration 600, loss = 2.0790
Checking accuracy on the val set
Got 436 / 1000 correct (43.60%)

Iteration 700, loss = 1.6753
Checking accuracy on the val set
Got 415 / 1000 correct (41.50%)
```

4.0.8 BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU

3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

[39]:
```
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

################################################################################
# TODO: Initialize the parameters of a three-layer ConvNet.                    #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

hidden_layer_size = 4000
learning_rate = 1e-2

conv_w1 = random_weight((channel_1,3, 5,5))
conv_b1 = zero_weight((channel_1))
conv_w2 = random_weight((channel_2,channel_1, 3,3))
conv_b2 = zero_weight((channel_2))
fc_w = random_weight((channel_2*32*32,10))
fc_b = zero_weight((10))



# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

```
Iteration 0, loss = 2.6227
Checking accuracy on the val set
```

```
Got 130 / 1000 correct (13.00%)

Iteration 100, loss = 1.6664
Checking accuracy on the val set
Got 409 / 1000 correct (40.90%)

Iteration 200, loss = 1.7540
Checking accuracy on the val set
Got 452 / 1000 correct (45.20%)

Iteration 300, loss = 1.4717
Checking accuracy on the val set
Got 458 / 1000 correct (45.80%)

Iteration 400, loss = 1.5065
Checking accuracy on the val set
Got 461 / 1000 correct (46.10%)

Iteration 500, loss = 1.5762
Checking accuracy on the val set
Got 506 / 1000 correct (50.60%)

Iteration 600, loss = 1.2786
Checking accuracy on the val set
Got 518 / 1000 correct (51.80%)

Iteration 700, loss = 1.3510
Checking accuracy on the val set
Got 525 / 1000 correct (52.50%)
```

5 Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the doc for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable

parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!

3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

5.0.1 Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

```
[40]: class TwoLayerFC(nn.Module):
          def __init__(self, input_size, hidden_size, num_classes):
              super().__init__()
              # assign layer objects to class attributes
              self.fc1 = nn.Linear(input_size, hidden_size)
              # nn.init package contains convenient initialization methods
              # http://pytorch.org/docs/master/nn.html#torch-nn-init
              nn.init.kaiming_normal_(self.fc1.weight)
              self.fc2 = nn.Linear(hidden_size, num_classes)
              nn.init.kaiming_normal_(self.fc2.weight)

          def forward(self, x):
              # forward always defines connectivity
              x = flatten(x)
              scores = self.fc2(F.relu(self.fc1(x)))
              return scores

      def test_TwoLayerFC():
          input_size = 50
          x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64,␣
       ↪feature dimension 50
          model = TwoLayerFC(input_size, 42, 10)
          scores = model(x)
          print(scores.size())  # you should see [64, 10]
      test_TwoLayerFC()
```

```
torch.Size([64, 10])
```

5.0.2 Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2

2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: http://pytorch.org/docs/stable/nn.html#conv2d

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 10)` for the shape of the output scores.

[41]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
        # architecture defined above.                                          #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.conv1 = nn.
 ↪Conv2d(in_channels=in_channel,out_channels=channel_1,kernel_size=(5,5),padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)

        self.conv2 = nn.
 ↪Conv2d(in_channels=channel_1,out_channels=channel_2,kernel_size=(3,3),padding=1)
        nn.init.kaiming_normal_(self.conv2.weight)

        self.fc = nn.Linear(channel_2*32*32,num_classes)
        nn.init.kaiming_normal_(self.fc.weight)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
        #                          END OF YOUR CODE                           ␣
 ↪#
        ########################################################################

    def forward(self, x):
        scores = None
        ########################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you      #
        # should use the layers you defined in __init__ and specify the        #
        # connectivity of those layers in forward()                            #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        scores = self.fc(flatten(F.relu(self.conv2(F.relu(self.conv1(x))))))
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
        #                             END OF YOUR CODE                         #
        ########################################################################
        return scores


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
 ↪size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8,␣
 ↪num_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 10])
```

5.0.3 Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

[42]:
```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 *␣
 ↪acc))
```

5.0.4 Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.

[43]:
```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train␣
 ↪for

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the␣
 ↪optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

5.0.5 Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

You don't need to tune any hyperparameters, but you should see model accuracies above 40% after training for one epoch.

[44]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.5098
Checking accuracy on validation set
Got 193 / 1000 correct (19.30)

Iteration 100, loss = 2.1788
Checking accuracy on validation set
Got 349 / 1000 correct (34.90)

Iteration 200, loss = 1.6492
Checking accuracy on validation set
Got 348 / 1000 correct (34.80)

Iteration 300, loss = 2.0876
Checking accuracy on validation set
Got 372 / 1000 correct (37.20)

Iteration 400, loss = 1.5418
Checking accuracy on validation set
Got 419 / 1000 correct (41.90)

Iteration 500, loss = 1.4774
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)

Iteration 600, loss = 1.9208
Checking accuracy on validation set
Got 373 / 1000 correct (37.30)

Iteration 700, loss = 1.6410
Checking accuracy on validation set
```

```
Got 429 / 1000 correct (42.90)
```

5.0.6 Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve above above 45% after training for one epoch.

You should train the model using stochastic gradient descent without momentum.

[45]:
```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

model =␣
 ↪ThreeLayerConvNet(3,channel_1=channel_1,channel_2=channel_2,num_classes=10)
optimizer = optim.SGD(model.parameters(),lr=learning_rate)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.1374
Checking accuracy on validation set
Got 113 / 1000 correct (11.30)

Iteration 100, loss = 1.8645
Checking accuracy on validation set
Got 368 / 1000 correct (36.80)

Iteration 200, loss = 1.8446
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 300, loss = 1.7545
Checking accuracy on validation set
Got 439 / 1000 correct (43.90)
```

```
Iteration 400, loss = 1.6938
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Iteration 500, loss = 1.5526
Checking accuracy on validation set
Got 464 / 1000 correct (46.40)

Iteration 600, loss = 1.6336
Checking accuracy on validation set
Got 503 / 1000 correct (50.30)

Iteration 700, loss = 1.4706
Checking accuracy on validation set
Got 502 / 1000 correct (50.20)
```

6 Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

6.0.1 Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

[46]:
```python
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                     momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.4166
Checking accuracy on validation set
Got 159 / 1000 correct (15.90)

Iteration 100, loss = 1.8929
Checking accuracy on validation set
Got 397 / 1000 correct (39.70)

Iteration 200, loss = 2.1851
Checking accuracy on validation set
Got 406 / 1000 correct (40.60)

Iteration 300, loss = 1.4090
Checking accuracy on validation set
Got 463 / 1000 correct (46.30)

Iteration 400, loss = 1.5593
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Iteration 500, loss = 1.6371
Checking accuracy on validation set
Got 426 / 1000 correct (42.60)

Iteration 600, loss = 2.0492
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Iteration 700, loss = 1.4094
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)
```

6.0.2 Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see accuracy above 55% after one epoch of training.

[47]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

################################################################################
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

model = nn.Sequential(
    nn.Conv2d(3,channel_1,(5,5),padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1,channel_2,(3,3),padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32,10)

)
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                     momentum=0.9, nesterov=True)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3090
Checking accuracy on validation set
Got 131 / 1000 correct (13.10)
```

```
Iteration 100, loss = 1.4657
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

Iteration 200, loss = 1.3461
Checking accuracy on validation set
Got 521 / 1000 correct (52.10)

Iteration 300, loss = 1.3089
Checking accuracy on validation set
Got 520 / 1000 correct (52.00)

Iteration 400, loss = 1.3651
Checking accuracy on validation set
Got 534 / 1000 correct (53.40)

Iteration 500, loss = 1.3361
Checking accuracy on validation set
Got 557 / 1000 correct (55.70)

Iteration 600, loss = 1.3330
Checking accuracy on validation set
Got 551 / 1000 correct (55.10)

Iteration 700, loss = 1.2802
Checking accuracy on validation set
Got 581 / 1000 correct (58.10)
```

7 Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

7.0.1 Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

7.0.2 Tips for training

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

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

7.0.3 Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview

7.0.4 Have fun and happy training!

[67]:
```
#returns a list with the correct nn.Modules for a single model using our␣
 ↪architecture
def␣
 ↪makeModel(in_shape,out_classes,N,K,M,numFilters,filterSizes,poolSizes,hiddenDims,convNorm=F
 ↪
    model = []
    #convolution and pooling
    for n in range(N):
        for k in range(K):

            model.append(nn.
 ↪Conv2d(in_shape[0],numFilters[n][k],(filterSizes[n][k],filterSizes[n][k]),padding␣
 ↪= filterSizes[n][k]//2))
            in_shape[0] = numFilters[n][k]
            if convNorm:
                model.append(nn.BatchNorm2d(in_shape[0]))
            model.append(nn.ReLU())
            if convDrop > 0:
                model.append(nn.Dropout2d(convDrop))
        model.append(nn.MaxPool2d(poolSizes[n]))
        in_shape[1] = 1 + ((in_shape[1]-poolSizes[n])//poolSizes[n])
        in_shape[2] = 1 + ((in_shape[2]-poolSizes[n])//poolSizes[n])


    #prep for affine layers
    model.append(Flatten())
    in_shape = in_shape[0]*in_shape[1]*in_shape[2]

    #affine layers
    for m in range(M-1):

        model.append(nn.Linear(in_shape,hiddenDims[m]))
        in_shape = hiddenDims[m]
        if affineNorm:
            model.append(nn.BatchNorm1d(in_shape))
        model.append(nn.ReLU())
        if affineDrop>0:
            model.append(nn.Dropout(affineDrop))


    #we always add an extra affine layer
    model.append(nn.Linear(in_shape,out_classes))
    """
    if affineNorm:
        model.append(nn.BatchNorm1d(out_classes))
    model.append(nn.ReLU())
    if affineDrop>0:
        model.append(nn.Dropout(affineDrop))
    """
    return model
```

[68]:
```
#similar to part34 but returns accuracy
def check_accuracy_part5(loader, model,verbose=0):

    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        if verbose>1:
            print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100␣
 ↪* acc))
        return acc


#similar to part34 but returns train and validation accuracy
def train_part5(model, optimizer,lossFunction=F.cross_entropy, epochs=1,␣
 ↪verbose = 0, print_every=100):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train␣
 ↪for

    Returns: train accuracy, validation accuracy
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        if verbose>1:
            print('')
            print("epoch:",e)
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = lossFunction(scores, y)

            # Zero out all of the gradients for the variables which the␣
↪optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                if verbose>1:
                    print('Iteration %d, loss = %.4f' % (t, loss.item()))
                    train_acc =␣
↪check_accuracy_part5(loader_train,model,verbose=verbose-1)
                    val_acc = check_accuracy_part5(loader_val,␣
↪model,verbose=verbose-1)
                    print('train accuracy: (%.2f), val accuracy: (%.2f)' % (100␣
↪* train_acc,100*val_acc))

    if verbose > 0:
        print('train loss = %.4f' % (loss.item()))
    train_acc = check_accuracy_part5(loader_train,model,verbose=verbose+1)
    val_acc = check_accuracy_part5(loader_val, model,verbose=verbose+1)

    return train_acc,val_acc
```

[69]:
```
params = [
    {'N': 2, 'K': 2, 'M': 3, 'numFilters': [[32, 32], [32, 32]], 'filterSizes':␣
↪[[5, 5], [5, 5]], 'poolSizes': [2, 2], 'hiddenDims': [256, 256, 256],␣
↪'convNorm': True, 'affineNorm': True, 'convDrop': 0, 'affineDrop': 0},
    {'N': 2, 'K': 3, 'M': 3, 'numFilters': [[32, 32, 32], [32, 32, 32]],␣
↪'filterSizes': [[5, 5, 5], [5, 5, 5]], 'poolSizes': [2, 2], 'hiddenDims':␣
↪[256, 256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop': 0,␣
↪'affineDrop': 0.1},
    {'N': 3, 'K': 2, 'M': 3, 'numFilters': [[32, 32], [32, 32],[32, 32]],␣
 ↪'filterSizes': [[5, 5], [5, 5], [5, 5]], 'poolSizes': [2, 2, 2],␣
 ↪'hiddenDims': [256, 256, 256], 'convNorm': True, 'affineNorm': True,␣
 ↪'convDrop': 0, 'affineDrop': 0.1},
    {'N': 3, 'K': 3, 'M': 3, 'numFilters': [[32, 32, 32], [32, 32, 32],[32, 32,␣
 ↪32]], 'filterSizes': [[5, 5, 5], [5, 5, 5], [5, 5, 5]], 'poolSizes': [2, 2,␣
 ↪2], 'hiddenDims': [256, 256, 256], 'convNorm': True, 'affineNorm': True,␣
 ↪'convDrop': 0, 'affineDrop': 0.1}
]
```

[70]:
```
#generate all the models
models = {}
for i in range(len(params)):
    print(params[i])
    models[i]= [makeModel([3,32,32],10,**params[i]),params[i]]
```

```
{'N': 2, 'K': 2, 'M': 3, 'numFilters': [[32, 32], [32, 32]], 'filterSizes': [[5,
5], [5, 5]], 'poolSizes': [2, 2], 'hiddenDims': [256, 256, 256], 'convNorm':
True, 'affineNorm': True, 'convDrop': 0, 'affineDrop': 0}
{'N': 2, 'K': 3, 'M': 3, 'numFilters': [[32, 32, 32], [32, 32, 32]],
'filterSizes': [[5, 5, 5], [5, 5, 5]], 'poolSizes': [2, 2], 'hiddenDims': [256,
256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop': 0, 'affineDrop':
0.1}
{'N': 3, 'K': 2, 'M': 3, 'numFilters': [[32, 32], [32, 32], [32, 32]],
'filterSizes': [[5, 5], [5, 5], [5, 5]], 'poolSizes': [2, 2, 2], 'hiddenDims':
[256, 256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop': 0,
'affineDrop': 0.1}
{'N': 3, 'K': 3, 'M': 3, 'numFilters': [[32, 32, 32], [32, 32, 32], [32, 32,
32]], 'filterSizes': [[5, 5, 5], [5, 5, 5], [5, 5, 5]], 'poolSizes': [2, 2, 2],
'hiddenDims': [256, 256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop':
0, 'affineDrop': 0.1}
```

[72]:
```
################################################################################
# TODO:                                                                        ␣
 ↪#
# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.      #
#                                                                              #
# Note that you can use the check_accuracy function to evaluate on either      #
# the test set or the validation set, by passing either loader_test or         #
# loader_val as the second argument to check_accuracy. You should not touch    #
# the test set until you have finished your architecture and  hyperparameter   #
# tuning, and only run the test set once at the end to report a final value.   #
################################################################################
model = None
optimizer = None
```

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

#we can check different optimizers
optimizers = {
    1:[optim.SGD,{"lr":3e-2,"momentum":0.9, "nesterov":True}]
              }

res = {}
best_val = -1
best_model = None

#iterate over all optimizers and models
#note,runtimes may be high since model is very deep
for k1,net in models.items():

    res[k1] = {}
    for k2,opt in optimizers.items():
        model = nn.Sequential(*net[0])
        optimizer = opt[0](model.parameters(),**opt[1])

        print(net[1])
        train_acc, val_acc = train_part5(model,␣
 ↪optimizer,epochs=5,verbose=2,print_every=500)
       ␣
 ↪print("--------------------------------------------------------------------")
        res[k1][k2] = [val_acc,train_acc,net[1]]
        #keep the best one
        if val_acc > best_val:
            best_val = val_acc
            best_model = model
            best_params = net[0]
            best_optimizer = optimizer


#finally set model and optimizer to be the best ones
model = nn.Sequential(*best_params)
optimizer = best_optimizer



# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE
################################################################################

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

```
{'N': 2, 'K': 2, 'M': 3, 'numFilters': [[32, 32], [32, 32]], 'filterSizes': [[5,
5], [5, 5]], 'poolSizes': [2, 2], 'hiddenDims': [256, 256, 256], 'convNorm':
True, 'affineNorm': True, 'convDrop': 0, 'affineDrop': 0}
epoch: 0
Iteration 0, loss = 11.7194
train accuracy: (9.99), val accuracy: (10.50)
Iteration 500, loss = 1.6614
train accuracy: (45.16), val accuracy: (47.30)
epoch: 1
Iteration 0, loss = 1.3787
train accuracy: (49.40), val accuracy: (49.00)
Iteration 500, loss = 1.1322
train accuracy: (54.88), val accuracy: (54.40)
epoch: 2
Iteration 0, loss = 1.1236
train accuracy: (57.61), val accuracy: (56.50)
Iteration 500, loss = 1.4296
train accuracy: (58.18), val accuracy: (55.70)
epoch: 3
Iteration 0, loss = 0.9580
train accuracy: (63.48), val accuracy: (63.10)
Iteration 500, loss = 0.9556
train accuracy: (65.53), val accuracy: (64.00)
epoch: 4
Iteration 0, loss = 0.9754
train accuracy: (65.99), val accuracy: (64.60)
Iteration 500, loss = 0.6796
train accuracy: (69.98), val accuracy: (65.10)
train loss = 1.3590
Got 34207 / 49000 correct (69.81)
Got 662 / 1000 correct (66.20)
--------------------------------------------------------------------
{'N': 2, 'K': 3, 'M': 3, 'numFilters': [[32, 32, 32], [32, 32, 32]],
'filterSizes': [[5, 5, 5], [5, 5, 5]], 'poolSizes': [2, 2], 'hiddenDims': [256,
256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop': 0, 'affineDrop':
0.1}
epoch: 0
Iteration 0, loss = 2.4090
train accuracy: (10.04), val accuracy: (7.90)
Iteration 500, loss = 1.3339
train accuracy: (55.45), val accuracy: (55.50)
epoch: 1
Iteration 0, loss = 1.1030
train accuracy: (63.47), val accuracy: (63.20)
Iteration 500, loss = 0.7344
train accuracy: (71.84), val accuracy: (71.30)
epoch: 2
Iteration 0, loss = 0.7638
```

```
train accuracy: (73.98), val accuracy: (72.00)
Iteration 500, loss = 0.6412
train accuracy: (76.82), val accuracy: (74.20)
epoch: 3
Iteration 0, loss = 0.6520
train accuracy: (79.09), val accuracy: (74.20)
Iteration 500, loss = 0.5458
train accuracy: (82.30), val accuracy: (75.60)
epoch: 4
Iteration 0, loss = 0.5135
train accuracy: (83.74), val accuracy: (79.50)
Iteration 500, loss = 0.4975
train accuracy: (80.42), val accuracy: (74.80)
train loss = 0.7808
Got 40614 / 49000 correct (82.89)
Got 761 / 1000 correct (76.10)
--------------------------------------------------------------------
{'N': 3, 'K': 2, 'M': 3, 'numFilters': [[32, 32], [32, 32], [32, 32]],
'filterSizes': [[5, 5], [5, 5], [5, 5]], 'poolSizes': [2, 2, 2], 'hiddenDims':
[256, 256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop': 0,
'affineDrop': 0.1}
epoch: 0
Iteration 0, loss = 2.4969
train accuracy: (9.96), val accuracy: (11.90)
Iteration 500, loss = 1.1590
train accuracy: (55.77), val accuracy: (58.50)
epoch: 1
Iteration 0, loss = 1.0063
train accuracy: (64.15), val accuracy: (62.40)
Iteration 500, loss = 1.0654
train accuracy: (68.23), val accuracy: (70.20)
epoch: 2
Iteration 0, loss = 0.9304
train accuracy: (72.16), val accuracy: (71.90)
Iteration 500, loss = 0.5867
train accuracy: (74.81), val accuracy: (72.00)
epoch: 3
Iteration 0, loss = 0.6000
train accuracy: (77.31), val accuracy: (74.70)
Iteration 500, loss = 0.5343
train accuracy: (80.50), val accuracy: (77.20)
epoch: 4
Iteration 0, loss = 0.4130
train accuracy: (80.10), val accuracy: (76.50)
Iteration 500, loss = 0.7158
train accuracy: (81.37), val accuracy: (74.90)
train loss = 0.5826
Got 41168 / 49000 correct (84.02)
Got 789 / 1000 correct (78.90)
--------------------------------------------------------------------
{'N': 3, 'K': 3, 'M': 3, 'numFilters': [[32, 32, 32], [32, 32, 32], [32, 32,
32]], 'filterSizes': [[5, 5, 5], [5, 5, 5], [5, 5, 5]], 'poolSizes': [2, 2, 2],
'hiddenDims': [256, 256, 256], 'convNorm': True, 'affineNorm': True, 'convDrop':
0, 'affineDrop': 0.1}
epoch: 0
Iteration 0, loss = 2.4615
train accuracy: (10.04), val accuracy: (7.90)
Iteration 500, loss = 1.5502
train accuracy: (46.35), val accuracy: (44.40)
epoch: 1
Iteration 0, loss = 1.2145
train accuracy: (52.06), val accuracy: (53.10)
Iteration 500, loss = 0.9970
train accuracy: (62.59), val accuracy: (63.00)
epoch: 2
Iteration 0, loss = 1.0972
train accuracy: (64.51), val accuracy: (61.80)
Iteration 500, loss = 0.7263
train accuracy: (64.88), val accuracy: (63.10)
epoch: 3
Iteration 0, loss = 1.0103
train accuracy: (74.45), val accuracy: (73.80)
Iteration 500, loss = 0.7460
train accuracy: (75.42), val accuracy: (74.50)
epoch: 4
Iteration 0, loss = 0.5229
train accuracy: (77.05), val accuracy: (74.90)
Iteration 500, loss = 0.8548
train accuracy: (80.80), val accuracy: (78.00)
train loss = 0.6792
Got 40061 / 49000 correct (81.76)
Got 803 / 1000 correct (80.30)
--------------------------------------------------------------------
Iteration 0, loss = 0.6771
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)

Iteration 100, loss = 0.6652
Checking accuracy on validation set
Got 818 / 1000 correct (81.80)

Iteration 200, loss = 0.6158
Checking accuracy on validation set
Got 767 / 1000 correct (76.70)

Iteration 300, loss = 0.6971
```

```
Checking accuracy on validation set
Got 783 / 1000 correct (78.30)

Iteration 400, loss = 0.4795
Checking accuracy on validation set
Got 793 / 1000 correct (79.30)

Iteration 500, loss = 0.5022
Checking accuracy on validation set
Got 784 / 1000 correct (78.40)

Iteration 600, loss = 0.5490
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 700, loss = 0.5482
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)

Iteration 0, loss = 0.3842
Checking accuracy on validation set
Got 774 / 1000 correct (77.40)

Iteration 100, loss = 0.4268
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 200, loss = 0.5409
Checking accuracy on validation set
Got 776 / 1000 correct (77.60)

Iteration 300, loss = 0.4927
Checking accuracy on validation set
Got 795 / 1000 correct (79.50)

Iteration 400, loss = 0.4949
Checking accuracy on validation set
Got 756 / 1000 correct (75.60)

Iteration 500, loss = 0.4661
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 600, loss = 0.7033
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 700, loss = 0.5735
```

```
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 0, loss = 0.3950
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)

Iteration 100, loss = 0.3750
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 200, loss = 0.6945
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 300, loss = 0.4205
Checking accuracy on validation set
Got 808 / 1000 correct (80.80)

Iteration 400, loss = 0.2925
Checking accuracy on validation set
Got 788 / 1000 correct (78.80)

Iteration 500, loss = 0.3789
Checking accuracy on validation set
Got 782 / 1000 correct (78.20)

Iteration 600, loss = 0.3635
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 700, loss = 0.5499
Checking accuracy on validation set
Got 803 / 1000 correct (80.30)

Iteration 0, loss = 0.3799
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 100, loss = 0.4211
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 200, loss = 0.3795
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 300, loss = 0.3012
```

```
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 400, loss = 0.3887
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 500, loss = 0.6740
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 600, loss = 0.4431
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 700, loss = 0.5459
Checking accuracy on validation set
Got 793 / 1000 correct (79.30)

Iteration 0, loss = 0.4799
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 100, loss = 0.3119
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 200, loss = 0.3127
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 300, loss = 0.5829
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)

Iteration 400, loss = 0.4230
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 500, loss = 0.5103
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)

Iteration 600, loss = 0.3377
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 700, loss = 0.3154
```

```
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 0, loss = 0.5207
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 100, loss = 0.4571
Checking accuracy on validation set
Got 788 / 1000 correct (78.80)

Iteration 200, loss = 0.5376
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 300, loss = 0.3596
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)

Iteration 400, loss = 0.3822
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)

Iteration 500, loss = 0.2954
Checking accuracy on validation set
Got 813 / 1000 correct (81.30)

Iteration 600, loss = 0.3253
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 700, loss = 0.3414
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 0, loss = 0.3367
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 100, loss = 0.1936
Checking accuracy on validation set
Got 820 / 1000 correct (82.00)

Iteration 200, loss = 0.3117
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 300, loss = 0.2120
```

```
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)

Iteration 400, loss = 0.3377
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 500, loss = 0.3791
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 600, loss = 0.4617
Checking accuracy on validation set
Got 813 / 1000 correct (81.30)

Iteration 700, loss = 0.2800
Checking accuracy on validation set
Got 826 / 1000 correct (82.60)

Iteration 0, loss = 0.3682
Checking accuracy on validation set
Got 830 / 1000 correct (83.00)

Iteration 100, loss = 0.2793
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 200, loss = 0.3253
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 300, loss = 0.2399
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 400, loss = 0.3432
Checking accuracy on validation set
Got 835 / 1000 correct (83.50)

Iteration 500, loss = 0.2734
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 600, loss = 0.2390
Checking accuracy on validation set
Got 834 / 1000 correct (83.40)

Iteration 700, loss = 0.3355
```

```
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 0, loss = 0.2295
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 100, loss = 0.1773
Checking accuracy on validation set
Got 814 / 1000 correct (81.40)

Iteration 200, loss = 0.2797
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 300, loss = 0.3776
Checking accuracy on validation set
Got 815 / 1000 correct (81.50)

Iteration 400, loss = 0.4293
Checking accuracy on validation set
Got 831 / 1000 correct (83.10)

Iteration 500, loss = 0.2573
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 600, loss = 0.3600
Checking accuracy on validation set
Got 814 / 1000 correct (81.40)

Iteration 700, loss = 0.3260
Checking accuracy on validation set
Got 835 / 1000 correct (83.50)

Iteration 0, loss = 0.3328
Checking accuracy on validation set
Got 827 / 1000 correct (82.70)

Iteration 100, loss = 0.2017
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 200, loss = 0.2173
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 300, loss = 0.2751
```

```
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 400, loss = 0.2934
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 500, loss = 0.2742
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 600, loss = 0.2966
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 700, loss = 0.4075
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)
```

7.1 Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

The architecture of my model is as follows

[[conv-[norm]-ReLu-[dropout]]x`K` -> pool]x`N` -> [affine-[norm]-ReLu-[dropout]]x(`M-1`) -> affine -> softmax

where norm and dropout are optional

When searching for the best `N`, `K` and `M` values I happened to find a very good parameters so i tried to fine tune my model around them. I also noticed that my model was overfitting a little bit so i tried adding dropout to the affine layers and increasing the value of `N`

7.2 Test set – run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy.

[73]:
```python
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 8215 / 10000 correct (82.15)
```

[53]:
```python

```