FullyConnectedNets

April 30, 2023

[3]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs231n/assignments/assignment2/'
# FOLDERNAME = None
# 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 cs231n/datasets/
# !bash get_datasets.sh
# %cd ..
# %cd ..
```

1 Multi-Layer Fully Connected Network

In this exercise, you will implement a fully connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the network initialization, forward pass, and backward pass. Throughout this assignment, you will be implementing layers in `cs231n/layers.py`. You can re-use your implementations for `affine_forward`, `affine_backward`, `relu_forward`, `relu_backward`, and `softmax_loss` from Assignment 1. For right now, don't worry about implementing dropout or batch/layer normalization yet, as you will add those features later.

[4]:
```
# Setup cell.
import time
import numpy as np
```

```
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

%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))))
```

[11]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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 Initial Loss and Gradient Check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. This is a good way to see if the initial losses seem reasonable.

For gradient checking, you should expect to see errors around 1e-7 or less.

[87]:
```
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 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
    )

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

    # Most of the errors should be on the order of e-7 or smaller.
    # NOTE: It is fine however to see an error for W2 on the order of e-5
    # for the check when reg = 0.0
    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(f"{name} relative error: {rel_error(grad_num, grads[name])}")
```

```
Running check with reg =  0
Initial loss:  3.3924281569462678
W1 relative error: 1.4168133462424317e-07
W2 relative error: 4.137539880876479e-07
b1 relative error: 4.315778531984205e-09
b2 relative error: 7.453008169246051e-10
Running check with reg =  3.14
Initial loss:  6.934574305719007
W1 relative error: 2.6844615261885408e-08
W2 relative error: 2.0930833914745013e-08
b1 relative error: 5.106545728293062e-08
b2 relative error: 1.697915685750258e-09
```

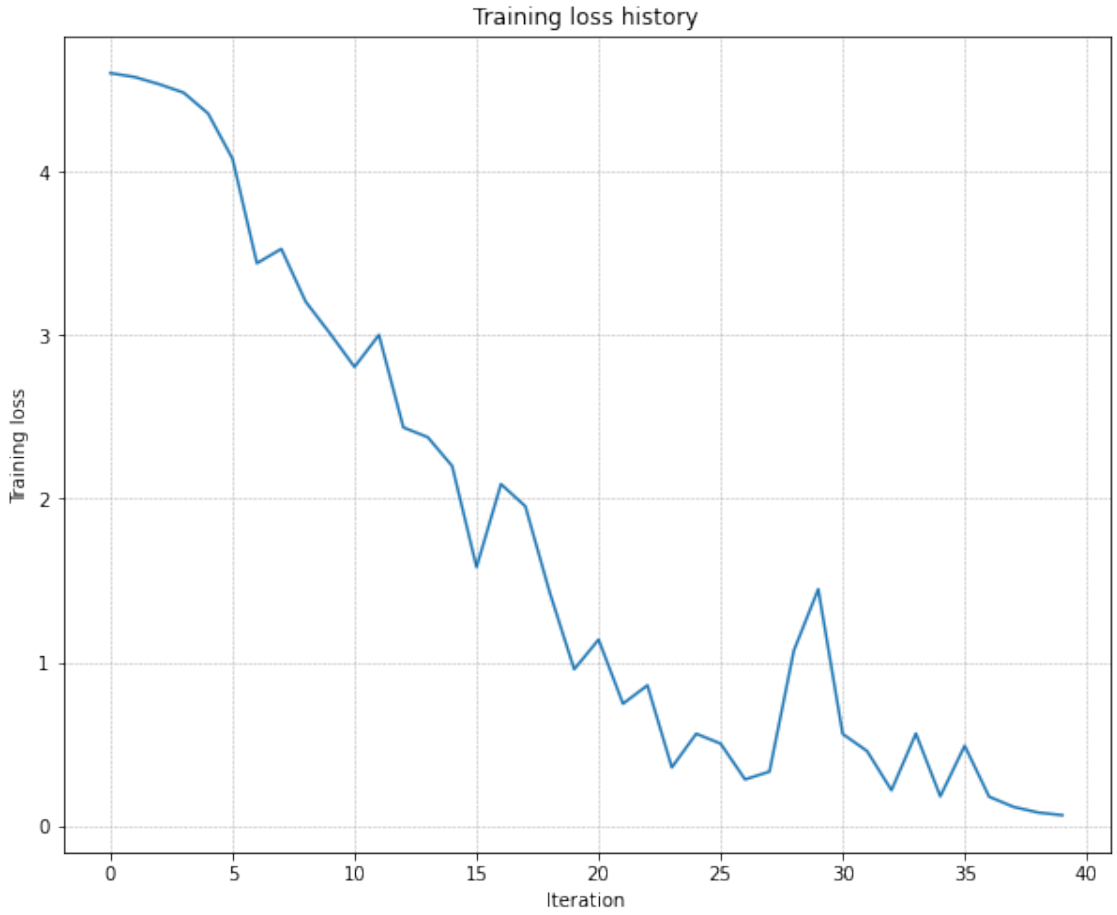
As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy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

[99]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
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 = 7e-4   # Experiment with this!
```

```
learning_rate = 2e-3  # Experiment with this!
model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule="sgd",
    optim_config={"learning_rate": learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 4.604725
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.082000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.123000
(Epoch 2 / 20) train acc: 0.400000; val_acc: 0.156000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.159000
(Epoch 4 / 20) train acc: 0.300000; val_acc: 0.161000
(Epoch 5 / 20) train acc: 0.280000; val_acc: 0.163000
(Iteration 11 / 40) loss: 2.806938
(Epoch 6 / 20) train acc: 0.380000; val_acc: 0.140000
(Epoch 7 / 20) train acc: 0.440000; val_acc: 0.129000
(Epoch 8 / 20) train acc: 0.580000; val_acc: 0.170000
(Epoch 9 / 20) train acc: 0.600000; val_acc: 0.164000
(Epoch 10 / 20) train acc: 0.680000; val_acc: 0.133000
(Iteration 21 / 40) loss: 1.138246
(Epoch 11 / 20) train acc: 0.720000; val_acc: 0.183000
(Epoch 12 / 20) train acc: 0.800000; val_acc: 0.179000
(Epoch 13 / 20) train acc: 0.860000; val_acc: 0.182000
(Epoch 14 / 20) train acc: 0.800000; val_acc: 0.178000
(Epoch 15 / 20) train acc: 0.920000; val_acc: 0.183000
(Iteration 31 / 40) loss: 0.561059
(Epoch 16 / 20) train acc: 0.940000; val_acc: 0.185000
(Epoch 17 / 20) train acc: 0.960000; val_acc: 0.180000
(Epoch 18 / 20) train acc: 0.960000; val_acc: 0.200000
```

```
(Epoch 19 / 20) train acc: 0.980000; val_acc: 0.180000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.189000
```



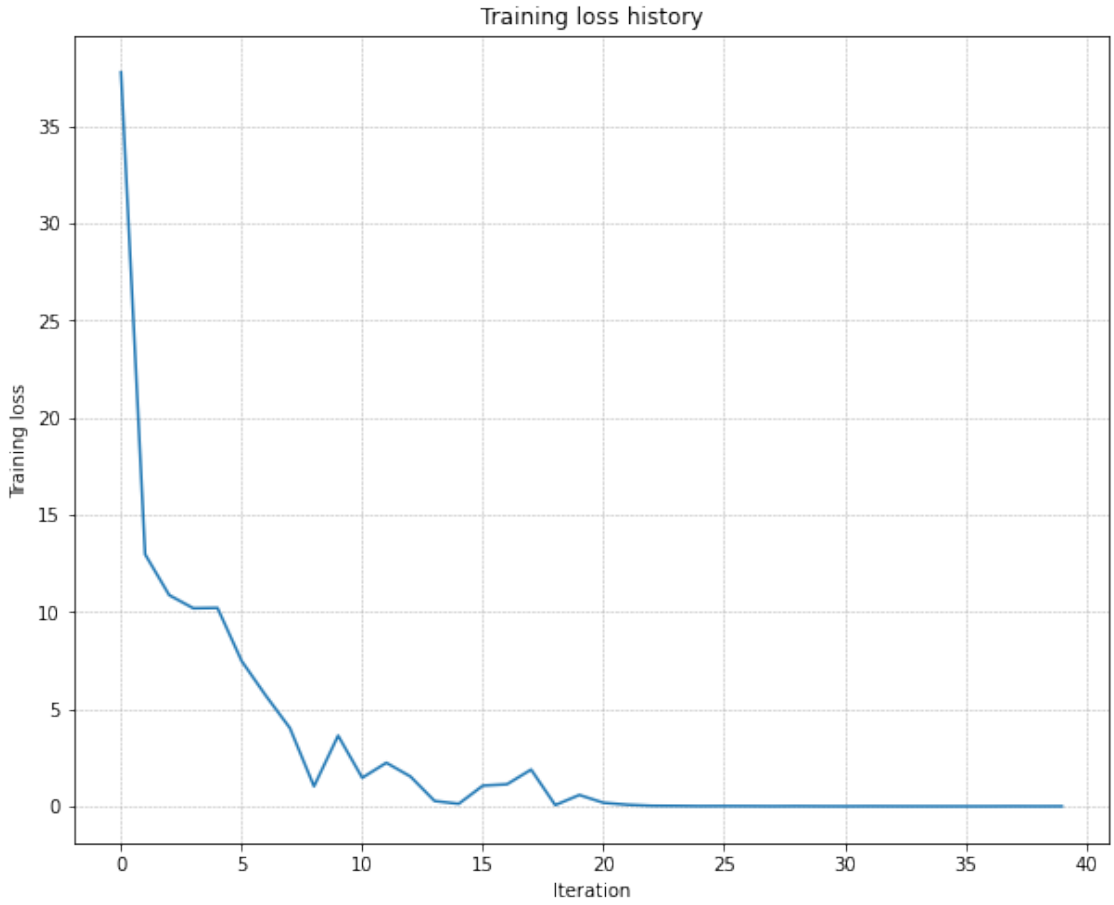
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[187]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
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 = 6e-2   # Experiment with this!
```

```
learning_rate = 2e-3  # Experiment with this!
model = FullyConnectedNet(
    [100, 100, 100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule='sgd',
    optim_config={'learning_rate': learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 37.782300
(Epoch 0 / 20) train acc: 0.080000; val_acc: 0.051000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.120000
(Epoch 2 / 20) train acc: 0.300000; val_acc: 0.093000
(Epoch 3 / 20) train acc: 0.460000; val_acc: 0.063000
(Epoch 4 / 20) train acc: 0.740000; val_acc: 0.083000
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.077000
(Iteration 11 / 40) loss: 1.480554
(Epoch 6 / 20) train acc: 0.820000; val_acc: 0.065000
(Epoch 7 / 20) train acc: 0.860000; val_acc: 0.080000
(Epoch 8 / 20) train acc: 0.900000; val_acc: 0.077000
(Epoch 9 / 20) train acc: 0.940000; val_acc: 0.069000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.074000
(Iteration 21 / 40) loss: 0.194569
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.082000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.083000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.087000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.089000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.091000
(Iteration 31 / 40) loss: 0.012004
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.090000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.089000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.092000
```

```
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.092000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.093000
```



1.2 Inline Question 1:

Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

1.3 Answer:

Training the 5-layer network was more difficult than training the 3-layer network. The 5-layer network was more sensitive to initialization scale. This can be explained by the bigger number of gradients and backpropagation steps in the 5-layer network that induces more calculation steps and thus more sensitivity to initialization scale. Small changes will be more amplified as the number of layers, and thus of steps, increases.

2 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.

2.1 SGD+Momentum

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

Open the file `cs231n/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.

[6]:
```
from cs231n.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
```

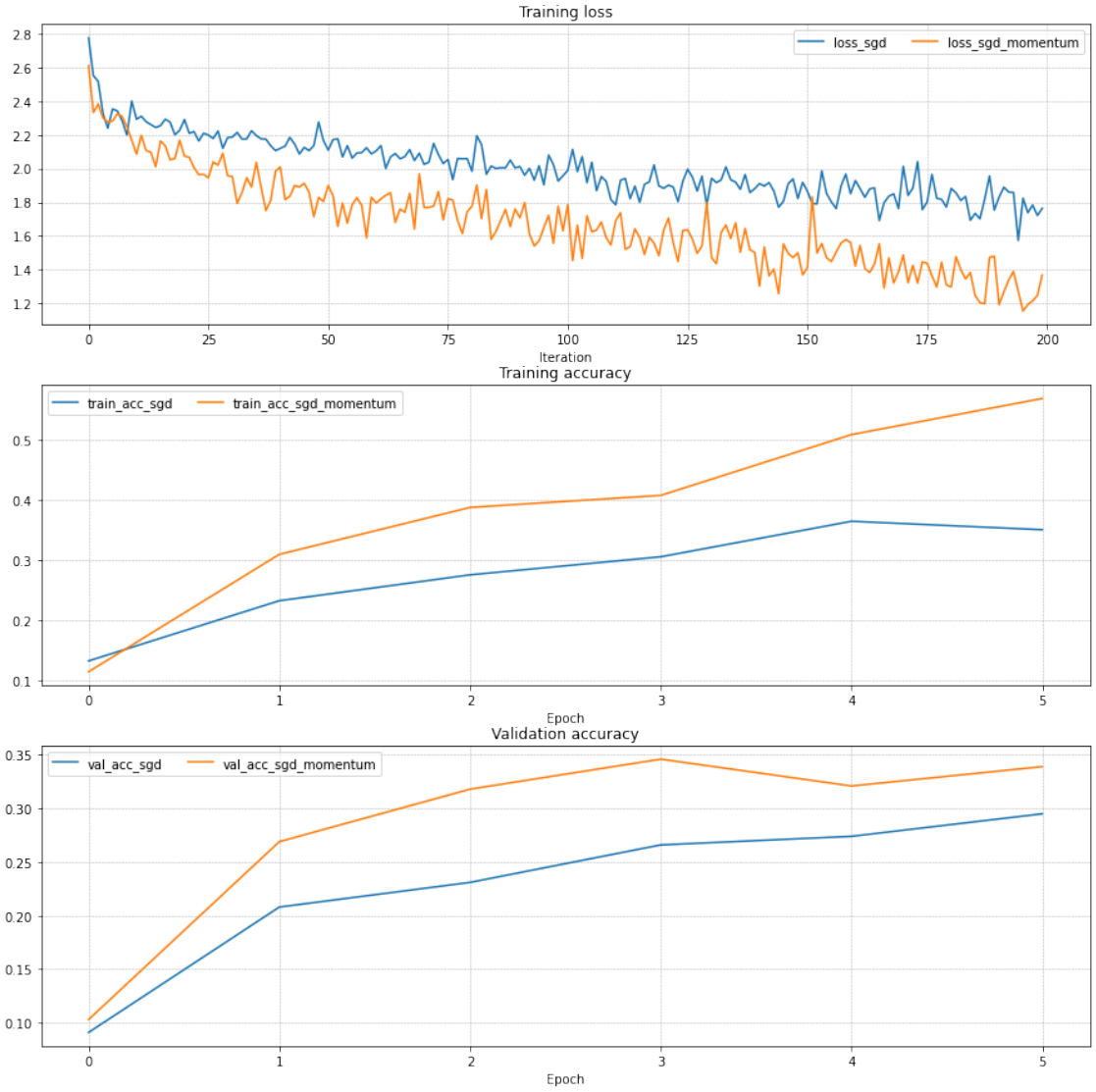
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[5]:
```
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()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"loss_{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"train_acc_{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"val_acc_{update_rule}")

for ax in axes:
    ax.legend(loc="best", ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  sgd
(Iteration 1 / 200) loss: 2.776360
(Epoch 0 / 5) train acc: 0.133000; val_acc: 0.091000
(Iteration 11 / 200) loss: 2.293176
(Iteration 21 / 200) loss: 2.291521
(Iteration 31 / 200) loss: 2.187513
(Epoch 1 / 5) train acc: 0.233000; val_acc: 0.208000
(Iteration 41 / 200) loss: 2.120019
(Iteration 51 / 200) loss: 2.110190
(Iteration 61 / 200) loss: 2.105030
(Iteration 71 / 200) loss: 2.025765
(Epoch 2 / 5) train acc: 0.276000; val_acc: 0.231000
(Iteration 81 / 200) loss: 1.984265
(Iteration 91 / 200) loss: 2.012653
(Iteration 101 / 200) loss: 1.990018
(Iteration 111 / 200) loss: 1.786057
(Epoch 3 / 5) train acc: 0.306000; val_acc: 0.266000
(Iteration 121 / 200) loss: 1.883376
(Iteration 131 / 200) loss: 1.943261
(Iteration 141 / 200) loss: 1.910895
(Iteration 151 / 200) loss: 1.865906
(Epoch 4 / 5) train acc: 0.365000; val_acc: 0.274000
(Iteration 161 / 200) loss: 1.929321
(Iteration 171 / 200) loss: 2.013536
(Iteration 181 / 200) loss: 1.882884
(Iteration 191 / 200) loss: 1.831260
(Epoch 5 / 5) train acc: 0.351000; val_acc: 0.295000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 2.611095
(Epoch 0 / 5) train acc: 0.115000; val_acc: 0.103000
(Iteration 11 / 200) loss: 2.085849
(Iteration 21 / 200) loss: 2.074134
(Iteration 31 / 200) loss: 1.952791
(Epoch 1 / 5) train acc: 0.310000; val_acc: 0.269000
(Iteration 41 / 200) loss: 2.009976
(Iteration 51 / 200) loss: 1.899888
(Iteration 61 / 200) loss: 1.795849
(Iteration 71 / 200) loss: 1.768942
(Epoch 2 / 5) train acc: 0.388000; val_acc: 0.318000
(Iteration 81 / 200) loss: 1.777336
(Iteration 91 / 200) loss: 1.708744
(Iteration 101 / 200) loss: 1.786629
(Iteration 111 / 200) loss: 1.692141
(Epoch 3 / 5) train acc: 0.408000; val_acc: 0.346000
(Iteration 121 / 200) loss: 1.633887
(Iteration 131 / 200) loss: 1.472320
(Iteration 141 / 200) loss: 1.303570
(Iteration 151 / 200) loss: 1.413389
```

```
(Epoch 4 / 5) train acc: 0.509000; val_acc: 0.321000
(Iteration 161 / 200) loss: 1.422052
(Iteration 171 / 200) loss: 1.487672
(Iteration 181 / 200) loss: 1.298038
(Iteration 191 / 200) loss: 1.192828
(Epoch 5 / 5) train acc: 0.569000; val_acc: 0.339000
```



2.2 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 `cs231n/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.

[8]:
```
# Test RMSProp implementation
from cs231n.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
```

[9]:
```
# Test Adam implementation
from cs231n.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
```

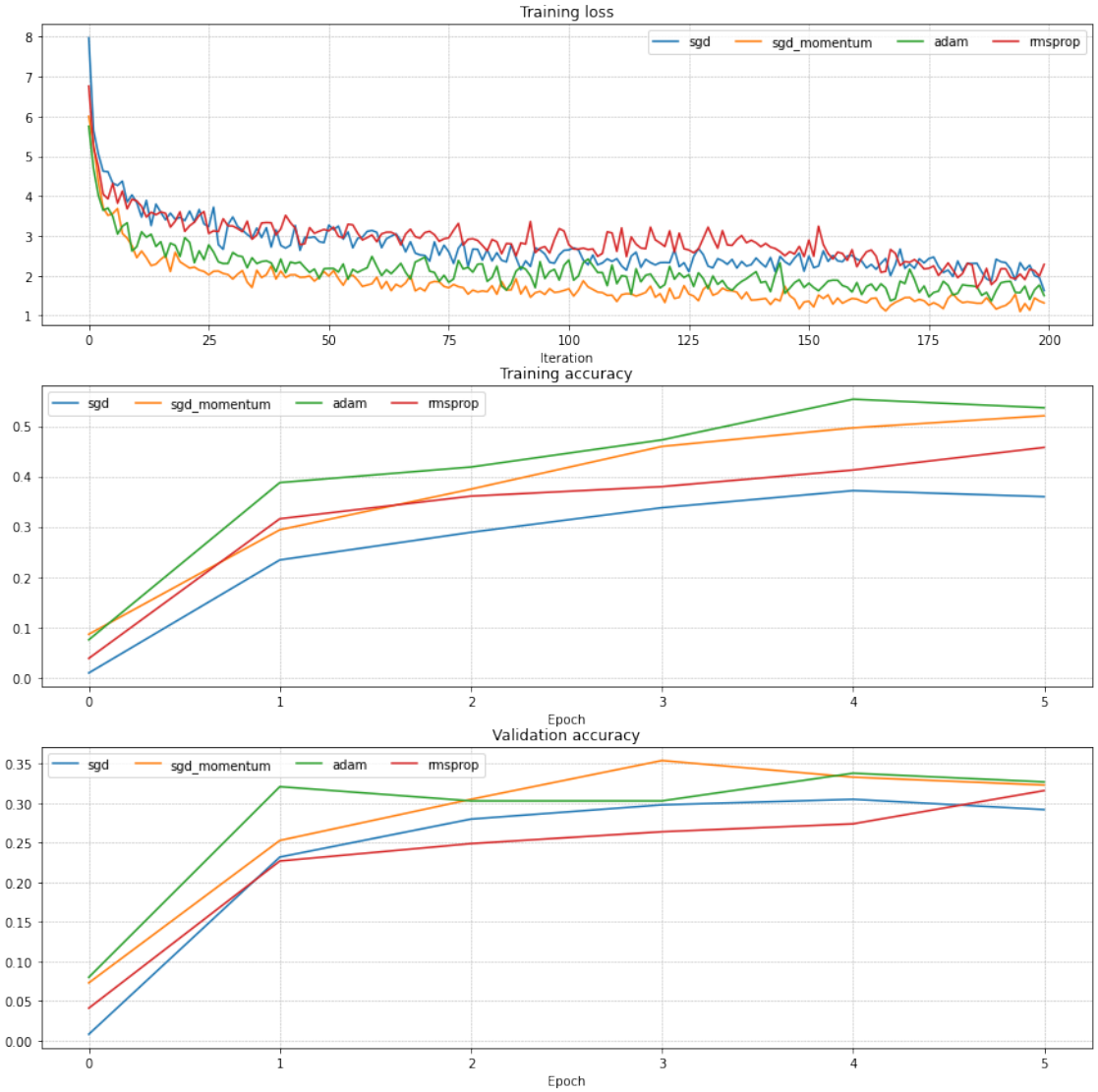
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[10]:
```
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()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"{update_rule}")

for ax in axes:
    ax.legend(loc='best', ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  adam
(Iteration 1 / 200) loss: 5.752198
(Epoch 0 / 5) train acc: 0.075000; val_acc: 0.080000
(Iteration 11 / 200) loss: 2.729689
(Iteration 21 / 200) loss: 2.956990
(Iteration 31 / 200) loss: 2.584068
(Epoch 1 / 5) train acc: 0.388000; val_acc: 0.321000
(Iteration 41 / 200) loss: 2.418603
(Iteration 51 / 200) loss: 2.181660
(Iteration 61 / 200) loss: 2.233033
(Iteration 71 / 200) loss: 2.467581
(Epoch 2 / 5) train acc: 0.419000; val_acc: 0.303000
(Iteration 81 / 200) loss: 2.088561
(Iteration 91 / 200) loss: 2.219185
(Iteration 101 / 200) loss: 2.395747
(Iteration 111 / 200) loss: 1.798579
(Epoch 3 / 5) train acc: 0.473000; val_acc: 0.303000
(Iteration 121 / 200) loss: 1.778342
(Iteration 131 / 200) loss: 1.992689
(Iteration 141 / 200) loss: 1.825535
(Iteration 151 / 200) loss: 1.812018
(Epoch 4 / 5) train acc: 0.554000; val_acc: 0.338000
(Iteration 161 / 200) loss: 1.829031
(Iteration 171 / 200) loss: 1.801084
(Iteration 181 / 200) loss: 1.517918
```

```
(Iteration 191 / 200) loss: 1.821995
(Epoch 5 / 5) train acc: 0.537000; val_acc: 0.327000

Running with  rmsprop
(Iteration 1 / 200) loss: 6.759055
(Epoch 0 / 5) train acc: 0.038000; val_acc: 0.041000
(Iteration 11 / 200) loss: 3.878140
(Iteration 21 / 200) loss: 3.114561
(Iteration 31 / 200) loss: 3.246890
(Epoch 1 / 5) train acc: 0.316000; val_acc: 0.227000
(Iteration 41 / 200) loss: 3.175140
(Iteration 51 / 200) loss: 3.126912
(Iteration 61 / 200) loss: 2.852072
(Iteration 71 / 200) loss: 3.083022
(Epoch 2 / 5) train acc: 0.361000; val_acc: 0.249000
(Iteration 81 / 200) loss: 2.928509
(Iteration 91 / 200) loss: 2.825300
(Iteration 101 / 200) loss: 2.777774
(Iteration 111 / 200) loss: 2.610797
(Epoch 3 / 5) train acc: 0.380000; val_acc: 0.264000
(Iteration 121 / 200) loss: 2.734270
(Iteration 131 / 200) loss: 2.940512
(Iteration 141 / 200) loss: 2.727979
(Iteration 151 / 200) loss: 2.894795
(Epoch 4 / 5) train acc: 0.413000; val_acc: 0.274000
(Iteration 161 / 200) loss: 2.224003
(Iteration 171 / 200) loss: 2.356149
(Iteration 181 / 200) loss: 2.144175
(Iteration 191 / 200) loss: 2.174863
(Epoch 5 / 5) train acc: 0.458000; val_acc: 0.316000
```



2.3 Inline Question 2:

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?

2.4 Answer:

With AdaGrad, the squares in the denominator increases as the training progresses. Therefore, the effective learning rate is decreasing. Then, the updates become smaller, and the learning becomes

slower.

Adam keeps track of both the first and second moments of the gradient and uses this information to compute adaptive learning rates for each parameter. Therefore, Adam does not have this issue of small updates, and thus of slow learning.

3 Train a Good Model!

Train the best fully connected 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 network.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional networks rather than fully connected networks.

Note: 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.

[14]:
```
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)*****

#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'],
}

# best learning_rate
lr = 5e-3
# best weight_scale
ws = 1e-2
# best normalization
normalization = 'batchnorm'
# best update_rule
update_rule = 'sgd'

# learning_rates = [1e-2, 5e-3, 2e-3, 1e-4]
# weight_scale = [5e-2, 1e-2, 5e-3, 1e-3]
```

```
# best_val_acc = -1

# for update_rule in ['sgd', 'sgd_momentum', 'adam', 'rmsprop']:
#     for normalization in [None, 'batchnorm', 'layernorm']:
#         for lr in learning_rates:
#             for ws in weight_scale:
print('Running with ' + update_rule + ', learning_rate = ' + str(lr) + ',␣
 ↪weight_scale = ' + str(ws), ', normalization = ' + str(normalization))
model = FullyConnectedNet(
    [100, 100, 100, 100, 100],
    weight_scale=ws,
    normalization=normalization,
)
solver = Solver(
    model,
    small_data,
    num_epochs=20,
    batch_size=100,
    update_rule=update_rule,
    optim_config={'learning_rate': lr},
    verbose=True
)
solver.train()

                # if solver.val_acc_history[-1] > best_val_acc:
                #     best_update_rule = update_rule
                #     best_normalization = normalization
                #     best_acc = solver.val_acc_history[-1]
                #     best_model = model
                #     best_solver = solver

best_model = model

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

# axes[0].plot(best_solver.loss_history, label=f"{best_update_rule,␣
 ↪best_normalization}")
# axes[1].plot(best_solver.train_acc_history, label=f"{best_update_rule,␣
 ↪best_normalization}")
```

```
# axes[2].plot(best_solver.val_acc_history, label=f"{best_update_rule,␣
 ↪best_normalization}")

axes[0].plot(solver.loss_history, label=f"{update_rule, normalization}")
axes[1].plot(solver.train_acc_history, label=f"{update_rule, normalization}")
axes[2].plot(solver.val_acc_history, label=f"{update_rule, normalization}")

for ax in axes:
    ax.legend(loc='best', ncol=1)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()

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

```
Running with sgd, learning_rate = 0.005, weight_scale = 0.01 , normalization =
batchnorm
(Iteration 1 / 9800) loss: 2.308289
(Epoch 0 / 20) train acc: 0.122000; val_acc: 0.124000
(Iteration 11 / 9800) loss: 2.282201
(Iteration 21 / 9800) loss: 2.277294
(Iteration 31 / 9800) loss: 2.258907
(Iteration 41 / 9800) loss: 2.240146
(Iteration 51 / 9800) loss: 2.236853
(Iteration 61 / 9800) loss: 2.200483
(Iteration 71 / 9800) loss: 2.145566
(Iteration 81 / 9800) loss: 2.108690
(Iteration 91 / 9800) loss: 2.080977
(Iteration 101 / 9800) loss: 2.085891
(Iteration 111 / 9800) loss: 2.063806
(Iteration 121 / 9800) loss: 2.044723
(Iteration 131 / 9800) loss: 1.989608
(Iteration 141 / 9800) loss: 1.986667
(Iteration 151 / 9800) loss: 2.026030
(Iteration 161 / 9800) loss: 1.968920
(Iteration 171 / 9800) loss: 1.905368
(Iteration 181 / 9800) loss: 1.963174
(Iteration 191 / 9800) loss: 1.949287
(Iteration 201 / 9800) loss: 1.907970
(Iteration 211 / 9800) loss: 1.891378
(Iteration 221 / 9800) loss: 1.850434
(Iteration 231 / 9800) loss: 1.826353
(Iteration 241 / 9800) loss: 1.919935
(Iteration 251 / 9800) loss: 1.776466
```

```
(Iteration 261 / 9800) loss: 1.875747
(Iteration 271 / 9800) loss: 1.844221
(Iteration 281 / 9800) loss: 1.935347
(Iteration 291 / 9800) loss: 1.778612
(Iteration 301 / 9800) loss: 1.864517
(Iteration 311 / 9800) loss: 1.861256
(Iteration 321 / 9800) loss: 1.788461
(Iteration 331 / 9800) loss: 1.825219
(Iteration 341 / 9800) loss: 1.671798
(Iteration 351 / 9800) loss: 1.836303
(Iteration 361 / 9800) loss: 1.723976
(Iteration 371 / 9800) loss: 1.763013
(Iteration 381 / 9800) loss: 1.657691
(Iteration 391 / 9800) loss: 1.731197
(Iteration 401 / 9800) loss: 1.702773
(Iteration 411 / 9800) loss: 1.751185
(Iteration 421 / 9800) loss: 1.692992
(Iteration 431 / 9800) loss: 1.704065
(Iteration 441 / 9800) loss: 1.684478
(Iteration 451 / 9800) loss: 1.742399
(Iteration 461 / 9800) loss: 1.640760
(Iteration 471 / 9800) loss: 1.718453
(Iteration 481 / 9800) loss: 1.896232
(Epoch 1 / 20) train acc: 0.416000; val_acc: 0.412000
(Iteration 491 / 9800) loss: 1.658214
(Iteration 501 / 9800) loss: 1.787903
(Iteration 511 / 9800) loss: 1.742867
(Iteration 521 / 9800) loss: 1.621359
(Iteration 531 / 9800) loss: 1.700530
(Iteration 541 / 9800) loss: 1.652953
(Iteration 551 / 9800) loss: 1.636601
(Iteration 561 / 9800) loss: 1.616825
(Iteration 571 / 9800) loss: 1.835541
(Iteration 581 / 9800) loss: 1.524691
(Iteration 591 / 9800) loss: 1.718854
(Iteration 601 / 9800) loss: 1.652140
(Iteration 611 / 9800) loss: 1.560507
(Iteration 621 / 9800) loss: 1.718896
(Iteration 631 / 9800) loss: 1.621317
(Iteration 641 / 9800) loss: 1.723254
(Iteration 651 / 9800) loss: 1.426551
(Iteration 661 / 9800) loss: 1.523225
(Iteration 671 / 9800) loss: 1.535787
(Iteration 681 / 9800) loss: 1.454170
(Iteration 691 / 9800) loss: 1.614155
(Iteration 701 / 9800) loss: 1.896978
(Iteration 711 / 9800) loss: 1.296324
(Iteration 721 / 9800) loss: 1.520639
```

```
(Iteration 731 / 9800) loss: 1.656467
(Iteration 741 / 9800) loss: 1.476072
(Iteration 751 / 9800) loss: 1.450802
(Iteration 761 / 9800) loss: 1.503230
(Iteration 771 / 9800) loss: 1.630655
(Iteration 781 / 9800) loss: 1.546725
(Iteration 791 / 9800) loss: 1.481948
(Iteration 801 / 9800) loss: 1.475325
(Iteration 811 / 9800) loss: 1.588596
(Iteration 821 / 9800) loss: 1.494431
(Iteration 831 / 9800) loss: 1.549809
(Iteration 841 / 9800) loss: 1.555539
(Iteration 851 / 9800) loss: 1.534892
(Iteration 861 / 9800) loss: 1.622303
(Iteration 871 / 9800) loss: 1.547734
(Iteration 881 / 9800) loss: 1.498438
(Iteration 891 / 9800) loss: 1.426499
(Iteration 901 / 9800) loss: 1.552759
(Iteration 911 / 9800) loss: 1.543741
(Iteration 921 / 9800) loss: 1.634369
(Iteration 931 / 9800) loss: 1.387971
(Iteration 941 / 9800) loss: 1.529694
(Iteration 951 / 9800) loss: 1.319405
(Iteration 961 / 9800) loss: 1.504389
(Iteration 971 / 9800) loss: 1.421561
(Epoch 2 / 20) train acc: 0.461000; val_acc: 0.458000
(Iteration 981 / 9800) loss: 1.433759
(Iteration 991 / 9800) loss: 1.508531
(Iteration 1001 / 9800) loss: 1.446544
(Iteration 1011 / 9800) loss: 1.580028
(Iteration 1021 / 9800) loss: 1.578192
(Iteration 1031 / 9800) loss: 1.368637
(Iteration 1041 / 9800) loss: 1.420318
(Iteration 1051 / 9800) loss: 1.501061
(Iteration 1061 / 9800) loss: 1.342488
(Iteration 1071 / 9800) loss: 1.426598
(Iteration 1081 / 9800) loss: 1.401390
(Iteration 1091 / 9800) loss: 1.466816
(Iteration 1101 / 9800) loss: 1.466989
(Iteration 1111 / 9800) loss: 1.343649
(Iteration 1121 / 9800) loss: 1.422316
(Iteration 1131 / 9800) loss: 1.387401
(Iteration 1141 / 9800) loss: 1.488189
(Iteration 1151 / 9800) loss: 1.662883
(Iteration 1161 / 9800) loss: 1.713398
(Iteration 1171 / 9800) loss: 1.434327
(Iteration 1181 / 9800) loss: 1.522623
(Iteration 1191 / 9800) loss: 1.345585
```

```
(Iteration 1201 / 9800) loss: 1.478734
(Iteration 1211 / 9800) loss: 1.583070
(Iteration 1221 / 9800) loss: 1.457697
(Iteration 1231 / 9800) loss: 1.335533
(Iteration 1241 / 9800) loss: 1.487380
(Iteration 1251 / 9800) loss: 1.286740
(Iteration 1261 / 9800) loss: 1.500654
(Iteration 1271 / 9800) loss: 1.540874
(Iteration 1281 / 9800) loss: 1.501559
(Iteration 1291 / 9800) loss: 1.365176
(Iteration 1301 / 9800) loss: 1.587036
(Iteration 1311 / 9800) loss: 1.720471
(Iteration 1321 / 9800) loss: 1.594854
(Iteration 1331 / 9800) loss: 1.446071
(Iteration 1341 / 9800) loss: 1.384616
(Iteration 1351 / 9800) loss: 1.380901
(Iteration 1361 / 9800) loss: 1.516054
(Iteration 1371 / 9800) loss: 1.466701
(Iteration 1381 / 9800) loss: 1.307786
(Iteration 1391 / 9800) loss: 1.570282
(Iteration 1401 / 9800) loss: 1.434032
(Iteration 1411 / 9800) loss: 1.370172
(Iteration 1421 / 9800) loss: 1.311272
(Iteration 1431 / 9800) loss: 1.392070
(Iteration 1441 / 9800) loss: 1.720184
(Iteration 1451 / 9800) loss: 1.328942
(Iteration 1461 / 9800) loss: 1.431920
(Epoch 3 / 20) train acc: 0.532000; val_acc: 0.484000
(Iteration 1471 / 9800) loss: 1.389300
(Iteration 1481 / 9800) loss: 1.390235
(Iteration 1491 / 9800) loss: 1.293109
(Iteration 1501 / 9800) loss: 1.344979
(Iteration 1511 / 9800) loss: 1.393801
(Iteration 1521 / 9800) loss: 1.399576
(Iteration 1531 / 9800) loss: 1.519894
(Iteration 1541 / 9800) loss: 1.291747
(Iteration 1551 / 9800) loss: 1.396597
(Iteration 1561 / 9800) loss: 1.354324
(Iteration 1571 / 9800) loss: 1.336859
(Iteration 1581 / 9800) loss: 1.494852
(Iteration 1591 / 9800) loss: 1.458557
(Iteration 1601 / 9800) loss: 1.236053
(Iteration 1611 / 9800) loss: 1.431615
(Iteration 1621 / 9800) loss: 1.335962
(Iteration 1631 / 9800) loss: 1.420261
(Iteration 1641 / 9800) loss: 1.395406
(Iteration 1651 / 9800) loss: 1.364202
(Iteration 1661 / 9800) loss: 1.467074
```

```
(Iteration 1671 / 9800) loss: 1.274326
(Iteration 1681 / 9800) loss: 1.296307
(Iteration 1691 / 9800) loss: 1.373483
(Iteration 1701 / 9800) loss: 1.287989
(Iteration 1711 / 9800) loss: 1.218765
(Iteration 1721 / 9800) loss: 1.308073
(Iteration 1731 / 9800) loss: 1.382893
(Iteration 1741 / 9800) loss: 1.232378
(Iteration 1751 / 9800) loss: 1.223898
(Iteration 1761 / 9800) loss: 1.531569
(Iteration 1771 / 9800) loss: 1.392981
(Iteration 1781 / 9800) loss: 1.508594
(Iteration 1791 / 9800) loss: 1.412225
(Iteration 1801 / 9800) loss: 1.229419
(Iteration 1811 / 9800) loss: 1.529901
(Iteration 1821 / 9800) loss: 1.269678
(Iteration 1831 / 9800) loss: 1.304651
(Iteration 1841 / 9800) loss: 1.357834
(Iteration 1851 / 9800) loss: 1.227394
(Iteration 1861 / 9800) loss: 1.241307
(Iteration 1871 / 9800) loss: 1.291757
(Iteration 1881 / 9800) loss: 1.391016
(Iteration 1891 / 9800) loss: 1.511494
(Iteration 1901 / 9800) loss: 1.422828
(Iteration 1911 / 9800) loss: 1.366802
(Iteration 1921 / 9800) loss: 1.407001
(Iteration 1931 / 9800) loss: 1.350639
(Iteration 1941 / 9800) loss: 1.385814
(Iteration 1951 / 9800) loss: 1.303947
(Epoch 4 / 20) train acc: 0.549000; val_acc: 0.504000
(Iteration 1961 / 9800) loss: 1.358053
(Iteration 1971 / 9800) loss: 1.449646
(Iteration 1981 / 9800) loss: 1.263687
(Iteration 1991 / 9800) loss: 1.239137
(Iteration 2001 / 9800) loss: 1.192117
(Iteration 2011 / 9800) loss: 1.161460
(Iteration 2021 / 9800) loss: 1.193711
(Iteration 2031 / 9800) loss: 1.206269
(Iteration 2041 / 9800) loss: 1.327432
(Iteration 2051 / 9800) loss: 1.161482
(Iteration 2061 / 9800) loss: 1.165330
(Iteration 2071 / 9800) loss: 1.183282
(Iteration 2081 / 9800) loss: 1.410398
(Iteration 2091 / 9800) loss: 1.320422
(Iteration 2101 / 9800) loss: 1.154473
(Iteration 2111 / 9800) loss: 1.305297
(Iteration 2121 / 9800) loss: 1.343199
(Iteration 2131 / 9800) loss: 1.315312
```

```
(Iteration 2141 / 9800) loss: 1.360216
(Iteration 2151 / 9800) loss: 1.380433
(Iteration 2161 / 9800) loss: 1.193666
(Iteration 2171 / 9800) loss: 1.197950
(Iteration 2181 / 9800) loss: 1.116634
(Iteration 2191 / 9800) loss: 1.114135
(Iteration 2201 / 9800) loss: 1.198305
(Iteration 2211 / 9800) loss: 1.294307
(Iteration 2221 / 9800) loss: 1.176907
(Iteration 2231 / 9800) loss: 1.271941
(Iteration 2241 / 9800) loss: 1.223481
(Iteration 2251 / 9800) loss: 1.469202
(Iteration 2261 / 9800) loss: 1.197438
(Iteration 2271 / 9800) loss: 1.328568
(Iteration 2281 / 9800) loss: 1.195305
(Iteration 2291 / 9800) loss: 1.187074
(Iteration 2301 / 9800) loss: 1.061341
(Iteration 2311 / 9800) loss: 1.287682
(Iteration 2321 / 9800) loss: 1.330854
(Iteration 2331 / 9800) loss: 1.225339
(Iteration 2341 / 9800) loss: 1.216505
(Iteration 2351 / 9800) loss: 1.284790
(Iteration 2361 / 9800) loss: 1.242270
(Iteration 2371 / 9800) loss: 1.238839
(Iteration 2381 / 9800) loss: 1.166274
(Iteration 2391 / 9800) loss: 1.132139
(Iteration 2401 / 9800) loss: 1.308653
(Iteration 2411 / 9800) loss: 1.220925
(Iteration 2421 / 9800) loss: 1.238057
(Iteration 2431 / 9800) loss: 1.310871
(Iteration 2441 / 9800) loss: 1.049786
(Epoch 5 / 20) train acc: 0.553000; val_acc: 0.547000
(Iteration 2451 / 9800) loss: 1.228226
(Iteration 2461 / 9800) loss: 1.207393
(Iteration 2471 / 9800) loss: 1.326927
(Iteration 2481 / 9800) loss: 1.173528
(Iteration 2491 / 9800) loss: 1.305568
(Iteration 2501 / 9800) loss: 1.157184
(Iteration 2511 / 9800) loss: 1.428277
(Iteration 2521 / 9800) loss: 1.319321
(Iteration 2531 / 9800) loss: 1.393432
(Iteration 2541 / 9800) loss: 1.297503
(Iteration 2551 / 9800) loss: 1.231319
(Iteration 2561 / 9800) loss: 1.240179
(Iteration 2571 / 9800) loss: 1.201236
(Iteration 2581 / 9800) loss: 1.288635
(Iteration 2591 / 9800) loss: 1.191765
(Iteration 2601 / 9800) loss: 1.258827
```

```
(Iteration 2611 / 9800) loss: 1.203554
(Iteration 2621 / 9800) loss: 1.304312
(Iteration 2631 / 9800) loss: 1.214340
(Iteration 2641 / 9800) loss: 1.144989
(Iteration 2651 / 9800) loss: 1.188122
(Iteration 2661 / 9800) loss: 1.119851
(Iteration 2671 / 9800) loss: 1.354352
(Iteration 2681 / 9800) loss: 1.290269
(Iteration 2691 / 9800) loss: 1.207436
(Iteration 2701 / 9800) loss: 1.337758
(Iteration 2711 / 9800) loss: 1.176359
(Iteration 2721 / 9800) loss: 1.429313
(Iteration 2731 / 9800) loss: 1.118385
(Iteration 2741 / 9800) loss: 1.300752
(Iteration 2751 / 9800) loss: 0.934643
(Iteration 2761 / 9800) loss: 1.420427
(Iteration 2771 / 9800) loss: 1.110417
(Iteration 2781 / 9800) loss: 1.160268
(Iteration 2791 / 9800) loss: 1.376139
(Iteration 2801 / 9800) loss: 1.157597
(Iteration 2811 / 9800) loss: 1.182622
(Iteration 2821 / 9800) loss: 1.306678
(Iteration 2831 / 9800) loss: 1.136374
(Iteration 2841 / 9800) loss: 1.181836
(Iteration 2851 / 9800) loss: 1.474932
(Iteration 2861 / 9800) loss: 1.141797
(Iteration 2871 / 9800) loss: 1.266682
(Iteration 2881 / 9800) loss: 1.315172
(Iteration 2891 / 9800) loss: 1.345064
(Iteration 2901 / 9800) loss: 1.158695
(Iteration 2911 / 9800) loss: 1.118147
(Iteration 2921 / 9800) loss: 1.234212
(Iteration 2931 / 9800) loss: 1.186623
(Epoch 6 / 20) train acc: 0.574000; val_acc: 0.541000
(Iteration 2941 / 9800) loss: 1.250554
(Iteration 2951 / 9800) loss: 1.266963
(Iteration 2961 / 9800) loss: 1.227096
(Iteration 2971 / 9800) loss: 1.197506
(Iteration 2981 / 9800) loss: 1.197780
(Iteration 2991 / 9800) loss: 1.279292
(Iteration 3001 / 9800) loss: 1.243963
(Iteration 3011 / 9800) loss: 1.064997
(Iteration 3021 / 9800) loss: 1.310956
(Iteration 3031 / 9800) loss: 1.274198
(Iteration 3041 / 9800) loss: 1.151450
(Iteration 3051 / 9800) loss: 1.119053
(Iteration 3061 / 9800) loss: 1.252148
(Iteration 3071 / 9800) loss: 1.166588
```

```
(Iteration 3081 / 9800) loss: 1.222801
(Iteration 3091 / 9800) loss: 1.160147
(Iteration 3101 / 9800) loss: 1.209968
(Iteration 3111 / 9800) loss: 1.202100
(Iteration 3121 / 9800) loss: 1.033491
(Iteration 3131 / 9800) loss: 1.267493
(Iteration 3141 / 9800) loss: 1.457315
(Iteration 3151 / 9800) loss: 1.022090
(Iteration 3161 / 9800) loss: 1.138662
(Iteration 3171 / 9800) loss: 1.246566
(Iteration 3181 / 9800) loss: 1.247337
(Iteration 3191 / 9800) loss: 0.995197
(Iteration 3201 / 9800) loss: 1.211937
(Iteration 3211 / 9800) loss: 1.088961
(Iteration 3221 / 9800) loss: 1.074348
(Iteration 3231 / 9800) loss: 1.254035
(Iteration 3241 / 9800) loss: 1.307819
(Iteration 3251 / 9800) loss: 1.182874
(Iteration 3261 / 9800) loss: 1.007037
(Iteration 3271 / 9800) loss: 1.109396
(Iteration 3281 / 9800) loss: 1.199330
(Iteration 3291 / 9800) loss: 1.260630
(Iteration 3301 / 9800) loss: 1.320001
(Iteration 3311 / 9800) loss: 1.313991
(Iteration 3321 / 9800) loss: 1.156682
(Iteration 3331 / 9800) loss: 1.267613
(Iteration 3341 / 9800) loss: 1.335196
(Iteration 3351 / 9800) loss: 1.020666
(Iteration 3361 / 9800) loss: 1.038882
(Iteration 3371 / 9800) loss: 1.079281
(Iteration 3381 / 9800) loss: 1.205071
(Iteration 3391 / 9800) loss: 1.063612
(Iteration 3401 / 9800) loss: 1.267779
(Iteration 3411 / 9800) loss: 1.164852
(Iteration 3421 / 9800) loss: 0.944869
(Epoch 7 / 20) train acc: 0.607000; val_acc: 0.529000
(Iteration 3431 / 9800) loss: 1.097254
(Iteration 3441 / 9800) loss: 1.300655
(Iteration 3451 / 9800) loss: 1.362087
(Iteration 3461 / 9800) loss: 1.089383
(Iteration 3471 / 9800) loss: 1.344977
(Iteration 3481 / 9800) loss: 1.029625
(Iteration 3491 / 9800) loss: 1.196229
(Iteration 3501 / 9800) loss: 1.160728
(Iteration 3511 / 9800) loss: 1.404624
(Iteration 3521 / 9800) loss: 1.066926
(Iteration 3531 / 9800) loss: 1.221129
(Iteration 3541 / 9800) loss: 1.116404
```

```
(Iteration 3551 / 9800) loss: 1.179769
(Iteration 3561 / 9800) loss: 1.275749
(Iteration 3571 / 9800) loss: 1.022379
(Iteration 3581 / 9800) loss: 1.243622
(Iteration 3591 / 9800) loss: 1.102602
(Iteration 3601 / 9800) loss: 0.949842
(Iteration 3611 / 9800) loss: 1.082757
(Iteration 3621 / 9800) loss: 1.210046
(Iteration 3631 / 9800) loss: 1.124616
(Iteration 3641 / 9800) loss: 1.347271
(Iteration 3651 / 9800) loss: 1.177986
(Iteration 3661 / 9800) loss: 0.917810
(Iteration 3671 / 9800) loss: 1.050448
(Iteration 3681 / 9800) loss: 1.139589
(Iteration 3691 / 9800) loss: 1.153858
(Iteration 3701 / 9800) loss: 1.045490
(Iteration 3711 / 9800) loss: 1.092212
(Iteration 3721 / 9800) loss: 1.246402
(Iteration 3731 / 9800) loss: 1.470130
(Iteration 3741 / 9800) loss: 1.116913
(Iteration 3751 / 9800) loss: 1.156716
(Iteration 3761 / 9800) loss: 1.166068
(Iteration 3771 / 9800) loss: 1.169580
(Iteration 3781 / 9800) loss: 0.926077
(Iteration 3791 / 9800) loss: 1.099943
(Iteration 3801 / 9800) loss: 1.115806
(Iteration 3811 / 9800) loss: 1.219507
(Iteration 3821 / 9800) loss: 1.145129
(Iteration 3831 / 9800) loss: 0.976609
(Iteration 3841 / 9800) loss: 1.156711
(Iteration 3851 / 9800) loss: 1.083604
(Iteration 3861 / 9800) loss: 1.026176
(Iteration 3871 / 9800) loss: 1.176912
(Iteration 3881 / 9800) loss: 1.140843
(Iteration 3891 / 9800) loss: 1.156572
(Iteration 3901 / 9800) loss: 1.284445
(Iteration 3911 / 9800) loss: 1.108717
(Epoch 8 / 20) train acc: 0.610000; val_acc: 0.534000
(Iteration 3921 / 9800) loss: 0.979426
(Iteration 3931 / 9800) loss: 1.332516
(Iteration 3941 / 9800) loss: 1.072678
(Iteration 3951 / 9800) loss: 1.049923
(Iteration 3961 / 9800) loss: 1.142221
(Iteration 3971 / 9800) loss: 1.159306
(Iteration 3981 / 9800) loss: 1.114376
(Iteration 3991 / 9800) loss: 1.288743
(Iteration 4001 / 9800) loss: 1.120362
(Iteration 4011 / 9800) loss: 1.240688
```

```
(Iteration 4021 / 9800) loss: 0.997900
(Iteration 4031 / 9800) loss: 1.135165
(Iteration 4041 / 9800) loss: 1.147609
(Iteration 4051 / 9800) loss: 1.320623
(Iteration 4061 / 9800) loss: 1.178183
(Iteration 4071 / 9800) loss: 1.100996
(Iteration 4081 / 9800) loss: 1.127679
(Iteration 4091 / 9800) loss: 1.147332
(Iteration 4101 / 9800) loss: 1.099201
(Iteration 4111 / 9800) loss: 1.042434
(Iteration 4121 / 9800) loss: 1.088984
(Iteration 4131 / 9800) loss: 1.116926
(Iteration 4141 / 9800) loss: 1.217044
(Iteration 4151 / 9800) loss: 1.041871
(Iteration 4161 / 9800) loss: 1.113987
(Iteration 4171 / 9800) loss: 0.978522
(Iteration 4181 / 9800) loss: 1.134514
(Iteration 4191 / 9800) loss: 1.074541
(Iteration 4201 / 9800) loss: 1.036323
(Iteration 4211 / 9800) loss: 1.074000
(Iteration 4221 / 9800) loss: 1.193464
(Iteration 4231 / 9800) loss: 1.051815
(Iteration 4241 / 9800) loss: 1.248775
(Iteration 4251 / 9800) loss: 1.019242
(Iteration 4261 / 9800) loss: 0.993401
(Iteration 4271 / 9800) loss: 1.211095
(Iteration 4281 / 9800) loss: 1.020432
(Iteration 4291 / 9800) loss: 1.250604
(Iteration 4301 / 9800) loss: 1.062609
(Iteration 4311 / 9800) loss: 1.275223
(Iteration 4321 / 9800) loss: 1.007489
(Iteration 4331 / 9800) loss: 1.177305
(Iteration 4341 / 9800) loss: 1.384400
(Iteration 4351 / 9800) loss: 1.077313
(Iteration 4361 / 9800) loss: 1.205438
(Iteration 4371 / 9800) loss: 1.134799
(Iteration 4381 / 9800) loss: 1.039326
(Iteration 4391 / 9800) loss: 1.084587
(Iteration 4401 / 9800) loss: 1.001100
(Epoch 9 / 20) train acc: 0.625000; val_acc: 0.539000
(Iteration 4411 / 9800) loss: 1.114103
(Iteration 4421 / 9800) loss: 1.092439
(Iteration 4431 / 9800) loss: 1.127770
(Iteration 4441 / 9800) loss: 1.236841
(Iteration 4451 / 9800) loss: 1.055241
(Iteration 4461 / 9800) loss: 0.932626
(Iteration 4471 / 9800) loss: 1.137735
(Iteration 4481 / 9800) loss: 1.111374
```

```
(Iteration 4491 / 9800) loss: 1.024382
(Iteration 4501 / 9800) loss: 1.306985
(Iteration 4511 / 9800) loss: 1.057358
(Iteration 4521 / 9800) loss: 1.035917
(Iteration 4531 / 9800) loss: 1.189153
(Iteration 4541 / 9800) loss: 1.119922
(Iteration 4551 / 9800) loss: 1.174912
(Iteration 4561 / 9800) loss: 1.115670
(Iteration 4571 / 9800) loss: 1.092002
(Iteration 4581 / 9800) loss: 1.033359
(Iteration 4591 / 9800) loss: 1.046763
(Iteration 4601 / 9800) loss: 1.028605
(Iteration 4611 / 9800) loss: 0.979767
(Iteration 4621 / 9800) loss: 1.072154
(Iteration 4631 / 9800) loss: 1.051326
(Iteration 4641 / 9800) loss: 1.204117
(Iteration 4651 / 9800) loss: 1.175562
(Iteration 4661 / 9800) loss: 1.077894
(Iteration 4671 / 9800) loss: 1.139229
(Iteration 4681 / 9800) loss: 1.070152
(Iteration 4691 / 9800) loss: 1.091763
(Iteration 4701 / 9800) loss: 1.044642
(Iteration 4711 / 9800) loss: 1.094462
(Iteration 4721 / 9800) loss: 1.135957
(Iteration 4731 / 9800) loss: 0.979853
(Iteration 4741 / 9800) loss: 1.019928
(Iteration 4751 / 9800) loss: 1.042979
(Iteration 4761 / 9800) loss: 1.042013
(Iteration 4771 / 9800) loss: 1.229600
(Iteration 4781 / 9800) loss: 1.043444
(Iteration 4791 / 9800) loss: 1.130852
(Iteration 4801 / 9800) loss: 0.926041
(Iteration 4811 / 9800) loss: 0.854010
(Iteration 4821 / 9800) loss: 0.940097
(Iteration 4831 / 9800) loss: 1.117685
(Iteration 4841 / 9800) loss: 0.778376
(Iteration 4851 / 9800) loss: 1.167204
(Iteration 4861 / 9800) loss: 0.896984
(Iteration 4871 / 9800) loss: 0.961666
(Iteration 4881 / 9800) loss: 1.249612
(Iteration 4891 / 9800) loss: 1.141776
(Epoch 10 / 20) train acc: 0.610000; val_acc: 0.547000
(Iteration 4901 / 9800) loss: 0.853189
(Iteration 4911 / 9800) loss: 1.048226
(Iteration 4921 / 9800) loss: 1.281939
(Iteration 4931 / 9800) loss: 0.952223
(Iteration 4941 / 9800) loss: 0.905501
(Iteration 4951 / 9800) loss: 1.307010
```

```
(Iteration 4961 / 9800) loss: 0.998570
(Iteration 4971 / 9800) loss: 0.989452
(Iteration 4981 / 9800) loss: 1.066523
(Iteration 4991 / 9800) loss: 1.021663
(Iteration 5001 / 9800) loss: 1.273234
(Iteration 5011 / 9800) loss: 1.114862
(Iteration 5021 / 9800) loss: 1.252974
(Iteration 5031 / 9800) loss: 0.957766
(Iteration 5041 / 9800) loss: 0.994247
(Iteration 5051 / 9800) loss: 1.109941
(Iteration 5061 / 9800) loss: 1.021961
(Iteration 5071 / 9800) loss: 1.114698
(Iteration 5081 / 9800) loss: 1.040364
(Iteration 5091 / 9800) loss: 0.829934
(Iteration 5101 / 9800) loss: 0.932718
(Iteration 5111 / 9800) loss: 1.202883
(Iteration 5121 / 9800) loss: 1.043451
(Iteration 5131 / 9800) loss: 0.999603
(Iteration 5141 / 9800) loss: 1.090553
(Iteration 5151 / 9800) loss: 1.240195
(Iteration 5161 / 9800) loss: 0.966358
(Iteration 5171 / 9800) loss: 0.875914
(Iteration 5181 / 9800) loss: 1.000026
(Iteration 5191 / 9800) loss: 1.156090
(Iteration 5201 / 9800) loss: 1.134303
(Iteration 5211 / 9800) loss: 0.890966
(Iteration 5221 / 9800) loss: 1.127795
(Iteration 5231 / 9800) loss: 1.046496
(Iteration 5241 / 9800) loss: 1.194736
(Iteration 5251 / 9800) loss: 1.184772
(Iteration 5261 / 9800) loss: 1.019628
(Iteration 5271 / 9800) loss: 0.869533
(Iteration 5281 / 9800) loss: 1.002842
(Iteration 5291 / 9800) loss: 0.928126
(Iteration 5301 / 9800) loss: 1.124616
(Iteration 5311 / 9800) loss: 1.071220
(Iteration 5321 / 9800) loss: 0.990526
(Iteration 5331 / 9800) loss: 1.069566
(Iteration 5341 / 9800) loss: 1.201049
(Iteration 5351 / 9800) loss: 1.212890
(Iteration 5361 / 9800) loss: 1.054603
(Iteration 5371 / 9800) loss: 1.116654
(Iteration 5381 / 9800) loss: 1.024270
(Epoch 11 / 20) train acc: 0.658000; val_acc: 0.552000
(Iteration 5391 / 9800) loss: 1.042879
(Iteration 5401 / 9800) loss: 0.927378
(Iteration 5411 / 9800) loss: 1.042198
(Iteration 5421 / 9800) loss: 1.077099
```

```
(Iteration 5431 / 9800) loss: 0.864745
(Iteration 5441 / 9800) loss: 0.907032
(Iteration 5451 / 9800) loss: 1.040711
(Iteration 5461 / 9800) loss: 1.036564
(Iteration 5471 / 9800) loss: 1.010390
(Iteration 5481 / 9800) loss: 1.215482
(Iteration 5491 / 9800) loss: 1.108193
(Iteration 5501 / 9800) loss: 0.836627
(Iteration 5511 / 9800) loss: 0.928139
(Iteration 5521 / 9800) loss: 1.215775
(Iteration 5531 / 9800) loss: 1.049472
(Iteration 5541 / 9800) loss: 0.934975
(Iteration 5551 / 9800) loss: 1.120967
(Iteration 5561 / 9800) loss: 1.070956
(Iteration 5571 / 9800) loss: 0.823801
(Iteration 5581 / 9800) loss: 1.121030
(Iteration 5591 / 9800) loss: 0.957771
(Iteration 5601 / 9800) loss: 1.065670
(Iteration 5611 / 9800) loss: 0.997617
(Iteration 5621 / 9800) loss: 1.142642
(Iteration 5631 / 9800) loss: 0.996304
(Iteration 5641 / 9800) loss: 1.179847
(Iteration 5651 / 9800) loss: 1.022452
(Iteration 5661 / 9800) loss: 1.119026
(Iteration 5671 / 9800) loss: 1.163676
(Iteration 5681 / 9800) loss: 1.111908
(Iteration 5691 / 9800) loss: 0.904548
(Iteration 5701 / 9800) loss: 1.112846
(Iteration 5711 / 9800) loss: 0.983969
(Iteration 5721 / 9800) loss: 0.922572
(Iteration 5731 / 9800) loss: 0.947866
(Iteration 5741 / 9800) loss: 1.045482
(Iteration 5751 / 9800) loss: 1.048958
(Iteration 5761 / 9800) loss: 1.054012
(Iteration 5771 / 9800) loss: 0.972227
(Iteration 5781 / 9800) loss: 0.981538
(Iteration 5791 / 9800) loss: 0.872637
(Iteration 5801 / 9800) loss: 0.873863
(Iteration 5811 / 9800) loss: 1.243909
(Iteration 5821 / 9800) loss: 0.931638
(Iteration 5831 / 9800) loss: 1.011560
(Iteration 5841 / 9800) loss: 1.024031
(Iteration 5851 / 9800) loss: 1.136636
(Iteration 5861 / 9800) loss: 0.840899
(Iteration 5871 / 9800) loss: 1.008129
(Epoch 12 / 20) train acc: 0.675000; val_acc: 0.551000
(Iteration 5881 / 9800) loss: 1.038121
(Iteration 5891 / 9800) loss: 1.191205
```

```
(Iteration 5901 / 9800) loss: 0.998718
(Iteration 5911 / 9800) loss: 1.077074
(Iteration 5921 / 9800) loss: 0.930199
(Iteration 5931 / 9800) loss: 1.148328
(Iteration 5941 / 9800) loss: 0.858926
(Iteration 5951 / 9800) loss: 1.038825
(Iteration 5961 / 9800) loss: 1.075548
(Iteration 5971 / 9800) loss: 1.024794
(Iteration 5981 / 9800) loss: 0.994058
(Iteration 5991 / 9800) loss: 1.014314
(Iteration 6001 / 9800) loss: 1.001016
(Iteration 6011 / 9800) loss: 1.027501
(Iteration 6021 / 9800) loss: 0.902265
(Iteration 6031 / 9800) loss: 0.979689
(Iteration 6041 / 9800) loss: 1.137364
(Iteration 6051 / 9800) loss: 0.996417
(Iteration 6061 / 9800) loss: 0.991973
(Iteration 6071 / 9800) loss: 1.241762
(Iteration 6081 / 9800) loss: 0.987468
(Iteration 6091 / 9800) loss: 0.807356
(Iteration 6101 / 9800) loss: 1.176460
(Iteration 6111 / 9800) loss: 0.936561
(Iteration 6121 / 9800) loss: 1.143583
(Iteration 6131 / 9800) loss: 1.210229
(Iteration 6141 / 9800) loss: 0.834147
(Iteration 6151 / 9800) loss: 1.034988
(Iteration 6161 / 9800) loss: 0.964298
(Iteration 6171 / 9800) loss: 1.026838
(Iteration 6181 / 9800) loss: 1.060710
(Iteration 6191 / 9800) loss: 1.150833
(Iteration 6201 / 9800) loss: 0.848329
(Iteration 6211 / 9800) loss: 0.973396
(Iteration 6221 / 9800) loss: 1.204178
(Iteration 6231 / 9800) loss: 0.883147
(Iteration 6241 / 9800) loss: 1.054132
(Iteration 6251 / 9800) loss: 0.969939
(Iteration 6261 / 9800) loss: 1.012935
(Iteration 6271 / 9800) loss: 0.955550
(Iteration 6281 / 9800) loss: 0.817970
(Iteration 6291 / 9800) loss: 1.075381
(Iteration 6301 / 9800) loss: 1.102507
(Iteration 6311 / 9800) loss: 0.885271
(Iteration 6321 / 9800) loss: 0.899592
(Iteration 6331 / 9800) loss: 1.042358
(Iteration 6341 / 9800) loss: 0.919144
(Iteration 6351 / 9800) loss: 1.010383
(Iteration 6361 / 9800) loss: 1.017594
(Epoch 13 / 20) train acc: 0.684000; val_acc: 0.545000
```

```
(Iteration 6371 / 9800) loss: 1.128917
(Iteration 6381 / 9800) loss: 0.969774
(Iteration 6391 / 9800) loss: 0.978187
(Iteration 6401 / 9800) loss: 1.047886
(Iteration 6411 / 9800) loss: 1.010715
(Iteration 6421 / 9800) loss: 1.090830
(Iteration 6431 / 9800) loss: 1.043650
(Iteration 6441 / 9800) loss: 1.096327
(Iteration 6451 / 9800) loss: 1.078829
(Iteration 6461 / 9800) loss: 1.082721
(Iteration 6471 / 9800) loss: 1.156634
(Iteration 6481 / 9800) loss: 1.014516
(Iteration 6491 / 9800) loss: 1.026985
(Iteration 6501 / 9800) loss: 0.972839
(Iteration 6511 / 9800) loss: 1.069643
(Iteration 6521 / 9800) loss: 1.084725
(Iteration 6531 / 9800) loss: 1.097310
(Iteration 6541 / 9800) loss: 0.901568
(Iteration 6551 / 9800) loss: 0.989646
(Iteration 6561 / 9800) loss: 0.979125
(Iteration 6571 / 9800) loss: 1.021033
(Iteration 6581 / 9800) loss: 1.023060
(Iteration 6591 / 9800) loss: 1.217657
(Iteration 6601 / 9800) loss: 0.973405
(Iteration 6611 / 9800) loss: 0.988331
(Iteration 6621 / 9800) loss: 0.883162
(Iteration 6631 / 9800) loss: 1.044839
(Iteration 6641 / 9800) loss: 1.026928
(Iteration 6651 / 9800) loss: 0.956450
(Iteration 6661 / 9800) loss: 0.856622
(Iteration 6671 / 9800) loss: 0.898637
(Iteration 6681 / 9800) loss: 0.880393
(Iteration 6691 / 9800) loss: 0.870056
(Iteration 6701 / 9800) loss: 1.027883
(Iteration 6711 / 9800) loss: 0.972125
(Iteration 6721 / 9800) loss: 0.684936
(Iteration 6731 / 9800) loss: 1.026463
(Iteration 6741 / 9800) loss: 0.940642
(Iteration 6751 / 9800) loss: 0.762445
(Iteration 6761 / 9800) loss: 0.844673
(Iteration 6771 / 9800) loss: 1.017973
(Iteration 6781 / 9800) loss: 1.228161
(Iteration 6791 / 9800) loss: 0.930179
(Iteration 6801 / 9800) loss: 0.904471
(Iteration 6811 / 9800) loss: 1.036600
(Iteration 6821 / 9800) loss: 0.913682
(Iteration 6831 / 9800) loss: 0.803517
(Iteration 6841 / 9800) loss: 0.761232
```

```
(Iteration 6851 / 9800) loss: 1.093553
(Epoch 14 / 20) train acc: 0.698000; val_acc: 0.533000
(Iteration 6861 / 9800) loss: 0.903612
(Iteration 6871 / 9800) loss: 0.954523
(Iteration 6881 / 9800) loss: 1.097000
(Iteration 6891 / 9800) loss: 0.967253
(Iteration 6901 / 9800) loss: 0.958749
(Iteration 6911 / 9800) loss: 0.976431
(Iteration 6921 / 9800) loss: 1.265678
(Iteration 6931 / 9800) loss: 0.965527
(Iteration 6941 / 9800) loss: 0.853428
(Iteration 6951 / 9800) loss: 1.073324
(Iteration 6961 / 9800) loss: 0.836950
(Iteration 6971 / 9800) loss: 0.931867
(Iteration 6981 / 9800) loss: 0.863315
(Iteration 6991 / 9800) loss: 0.774027
(Iteration 7001 / 9800) loss: 0.962478
(Iteration 7011 / 9800) loss: 1.116603
(Iteration 7021 / 9800) loss: 1.121348
(Iteration 7031 / 9800) loss: 0.875181
(Iteration 7041 / 9800) loss: 0.927286
(Iteration 7051 / 9800) loss: 0.984336
(Iteration 7061 / 9800) loss: 0.882478
(Iteration 7071 / 9800) loss: 1.014464
(Iteration 7081 / 9800) loss: 1.010193
(Iteration 7091 / 9800) loss: 1.049574
(Iteration 7101 / 9800) loss: 0.994789
(Iteration 7111 / 9800) loss: 1.044654
(Iteration 7121 / 9800) loss: 1.064008
(Iteration 7131 / 9800) loss: 1.052082
(Iteration 7141 / 9800) loss: 0.856600
(Iteration 7151 / 9800) loss: 0.959840
(Iteration 7161 / 9800) loss: 0.854941
(Iteration 7171 / 9800) loss: 0.716065
(Iteration 7181 / 9800) loss: 0.913461
(Iteration 7191 / 9800) loss: 0.908666
(Iteration 7201 / 9800) loss: 1.033007
(Iteration 7211 / 9800) loss: 0.801889
(Iteration 7221 / 9800) loss: 0.845586
(Iteration 7231 / 9800) loss: 0.899254
(Iteration 7241 / 9800) loss: 0.864499
(Iteration 7251 / 9800) loss: 0.878623
(Iteration 7261 / 9800) loss: 1.269730
(Iteration 7271 / 9800) loss: 0.970121
(Iteration 7281 / 9800) loss: 0.834341
(Iteration 7291 / 9800) loss: 0.902408
(Iteration 7301 / 9800) loss: 0.789627
(Iteration 7311 / 9800) loss: 0.974255
```

```
(Iteration 7321 / 9800) loss: 0.935987
(Iteration 7331 / 9800) loss: 0.875249
(Iteration 7341 / 9800) loss: 1.137663
(Epoch 15 / 20) train acc: 0.688000; val_acc: 0.562000
(Iteration 7351 / 9800) loss: 1.001409
(Iteration 7361 / 9800) loss: 1.095835
(Iteration 7371 / 9800) loss: 0.911448
(Iteration 7381 / 9800) loss: 0.731834
(Iteration 7391 / 9800) loss: 1.100130
(Iteration 7401 / 9800) loss: 0.939929
(Iteration 7411 / 9800) loss: 0.812998
(Iteration 7421 / 9800) loss: 0.863017
(Iteration 7431 / 9800) loss: 0.941642
(Iteration 7441 / 9800) loss: 0.975955
(Iteration 7451 / 9800) loss: 0.784585
(Iteration 7461 / 9800) loss: 0.980105
(Iteration 7471 / 9800) loss: 0.698085
(Iteration 7481 / 9800) loss: 0.794933
(Iteration 7491 / 9800) loss: 1.018391
(Iteration 7501 / 9800) loss: 0.749109
(Iteration 7511 / 9800) loss: 1.026564
(Iteration 7521 / 9800) loss: 1.209412
(Iteration 7531 / 9800) loss: 1.010056
(Iteration 7541 / 9800) loss: 0.986463
(Iteration 7551 / 9800) loss: 0.867182
(Iteration 7561 / 9800) loss: 0.826569
(Iteration 7571 / 9800) loss: 0.899819
(Iteration 7581 / 9800) loss: 0.927080
(Iteration 7591 / 9800) loss: 0.848694
(Iteration 7601 / 9800) loss: 0.888239
(Iteration 7611 / 9800) loss: 0.827361
(Iteration 7621 / 9800) loss: 0.972428
(Iteration 7631 / 9800) loss: 0.893340
(Iteration 7641 / 9800) loss: 0.884586
(Iteration 7651 / 9800) loss: 1.024706
(Iteration 7661 / 9800) loss: 1.045420
(Iteration 7671 / 9800) loss: 1.198505
(Iteration 7681 / 9800) loss: 0.939672
(Iteration 7691 / 9800) loss: 1.196586
(Iteration 7701 / 9800) loss: 0.856915
(Iteration 7711 / 9800) loss: 0.803178
(Iteration 7721 / 9800) loss: 0.931075
(Iteration 7731 / 9800) loss: 0.846213
(Iteration 7741 / 9800) loss: 0.973924
(Iteration 7751 / 9800) loss: 0.800830
(Iteration 7761 / 9800) loss: 1.049640
(Iteration 7771 / 9800) loss: 0.867421
(Iteration 7781 / 9800) loss: 0.855026
```

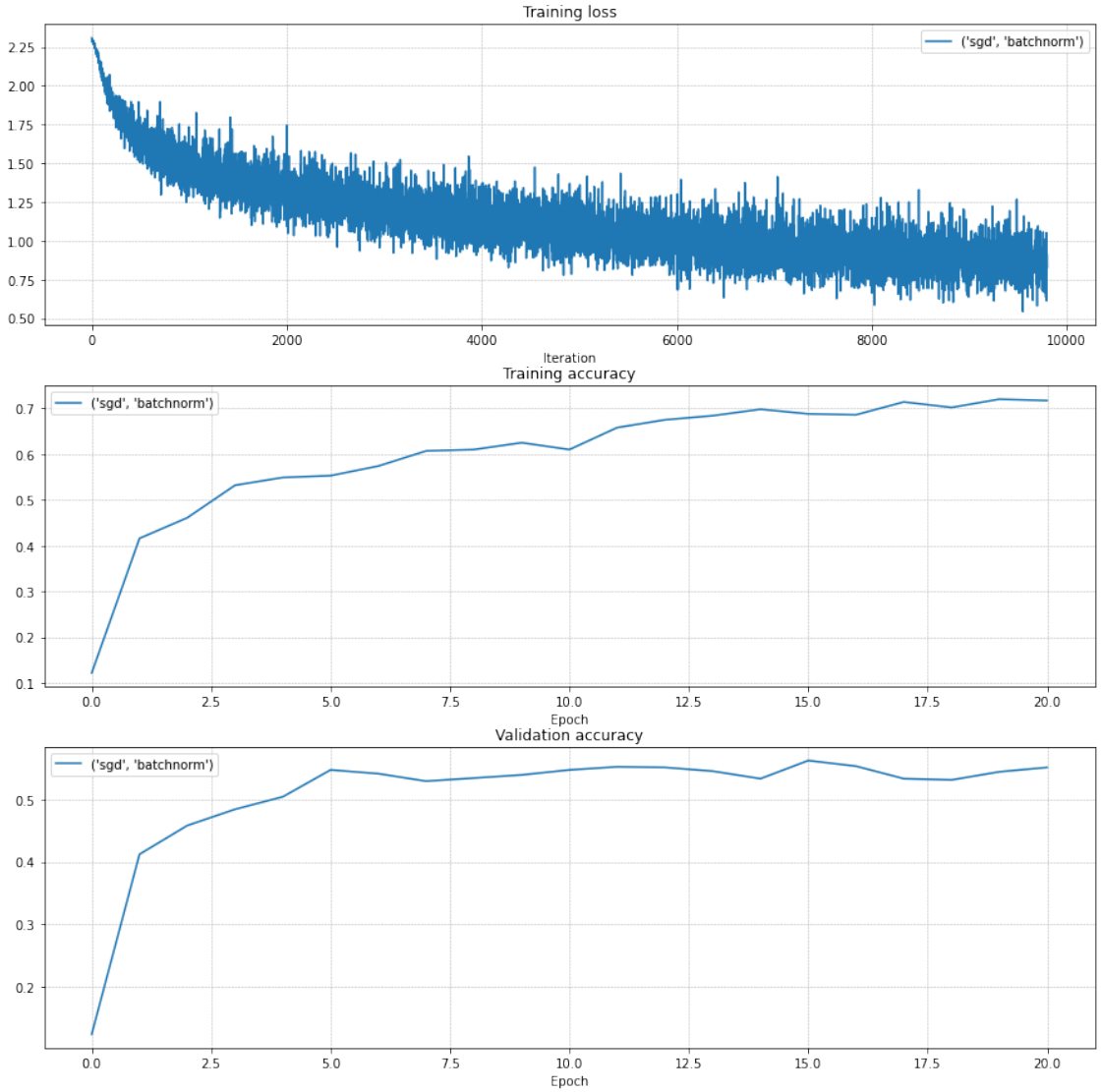
```
(Iteration 7791 / 9800) loss: 0.911755
(Iteration 7801 / 9800) loss: 0.936005
(Iteration 7811 / 9800) loss: 0.830656
(Iteration 7821 / 9800) loss: 0.874620
(Iteration 7831 / 9800) loss: 0.966927
(Epoch 16 / 20) train acc: 0.686000; val_acc: 0.553000
(Iteration 7841 / 9800) loss: 0.709133
(Iteration 7851 / 9800) loss: 1.057954
(Iteration 7861 / 9800) loss: 0.809761
(Iteration 7871 / 9800) loss: 1.096146
(Iteration 7881 / 9800) loss: 0.936006
(Iteration 7891 / 9800) loss: 0.895628
(Iteration 7901 / 9800) loss: 1.072401
(Iteration 7911 / 9800) loss: 0.876754
(Iteration 7921 / 9800) loss: 0.891471
(Iteration 7931 / 9800) loss: 0.796914
(Iteration 7941 / 9800) loss: 0.916161
(Iteration 7951 / 9800) loss: 0.858926
(Iteration 7961 / 9800) loss: 1.027783
(Iteration 7971 / 9800) loss: 0.860645
(Iteration 7981 / 9800) loss: 0.913543
(Iteration 7991 / 9800) loss: 0.966405
(Iteration 8001 / 9800) loss: 1.006843
(Iteration 8011 / 9800) loss: 0.675220
(Iteration 8021 / 9800) loss: 1.094731
(Iteration 8031 / 9800) loss: 0.985321
(Iteration 8041 / 9800) loss: 0.987216
(Iteration 8051 / 9800) loss: 0.962105
(Iteration 8061 / 9800) loss: 1.034606
(Iteration 8071 / 9800) loss: 1.020772
(Iteration 8081 / 9800) loss: 0.778934
(Iteration 8091 / 9800) loss: 0.732334
(Iteration 8101 / 9800) loss: 0.742898
(Iteration 8111 / 9800) loss: 1.188504
(Iteration 8121 / 9800) loss: 1.084790
(Iteration 8131 / 9800) loss: 0.912208
(Iteration 8141 / 9800) loss: 0.902791
(Iteration 8151 / 9800) loss: 0.744939
(Iteration 8161 / 9800) loss: 0.975201
(Iteration 8171 / 9800) loss: 0.771059
(Iteration 8181 / 9800) loss: 0.902774
(Iteration 8191 / 9800) loss: 0.787144
(Iteration 8201 / 9800) loss: 0.763560
(Iteration 8211 / 9800) loss: 0.794641
(Iteration 8221 / 9800) loss: 0.954660
(Iteration 8231 / 9800) loss: 1.079400
(Iteration 8241 / 9800) loss: 0.820947
(Iteration 8251 / 9800) loss: 0.898181
```

```
(Iteration 8261 / 9800) loss: 0.900396
(Iteration 8271 / 9800) loss: 0.655158
(Iteration 8281 / 9800) loss: 0.900797
(Iteration 8291 / 9800) loss: 0.819546
(Iteration 8301 / 9800) loss: 0.918205
(Iteration 8311 / 9800) loss: 0.977715
(Iteration 8321 / 9800) loss: 0.924724
(Epoch 17 / 20) train acc: 0.714000; val_acc: 0.533000
(Iteration 8331 / 9800) loss: 0.913825
(Iteration 8341 / 9800) loss: 0.833086
(Iteration 8351 / 9800) loss: 0.943220
(Iteration 8361 / 9800) loss: 1.009146
(Iteration 8371 / 9800) loss: 0.961553
(Iteration 8381 / 9800) loss: 0.891845
(Iteration 8391 / 9800) loss: 0.722746
(Iteration 8401 / 9800) loss: 0.958501
(Iteration 8411 / 9800) loss: 0.894421
(Iteration 8421 / 9800) loss: 0.971460
(Iteration 8431 / 9800) loss: 1.133161
(Iteration 8441 / 9800) loss: 0.899469
(Iteration 8451 / 9800) loss: 0.978942
(Iteration 8461 / 9800) loss: 0.794599
(Iteration 8471 / 9800) loss: 0.882685
(Iteration 8481 / 9800) loss: 0.821220
(Iteration 8491 / 9800) loss: 0.965053
(Iteration 8501 / 9800) loss: 1.064234
(Iteration 8511 / 9800) loss: 0.776081
(Iteration 8521 / 9800) loss: 0.887613
(Iteration 8531 / 9800) loss: 0.826798
(Iteration 8541 / 9800) loss: 0.907488
(Iteration 8551 / 9800) loss: 0.937812
(Iteration 8561 / 9800) loss: 0.752889
(Iteration 8571 / 9800) loss: 0.785657
(Iteration 8581 / 9800) loss: 0.987261
(Iteration 8591 / 9800) loss: 0.831909
(Iteration 8601 / 9800) loss: 0.906388
(Iteration 8611 / 9800) loss: 1.115934
(Iteration 8621 / 9800) loss: 1.043188
(Iteration 8631 / 9800) loss: 0.979509
(Iteration 8641 / 9800) loss: 0.882022
(Iteration 8651 / 9800) loss: 0.790055
(Iteration 8661 / 9800) loss: 0.920673
(Iteration 8671 / 9800) loss: 0.832287
(Iteration 8681 / 9800) loss: 1.101514
(Iteration 8691 / 9800) loss: 0.625249
(Iteration 8701 / 9800) loss: 1.065745
(Iteration 8711 / 9800) loss: 0.748349
(Iteration 8721 / 9800) loss: 0.798058
```

```
(Iteration 8731 / 9800) loss: 0.858706
(Iteration 8741 / 9800) loss: 0.879407
(Iteration 8751 / 9800) loss: 0.959441
(Iteration 8761 / 9800) loss: 0.866326
(Iteration 8771 / 9800) loss: 0.877291
(Iteration 8781 / 9800) loss: 0.918862
(Iteration 8791 / 9800) loss: 0.792474
(Iteration 8801 / 9800) loss: 1.036747
(Iteration 8811 / 9800) loss: 0.802539
(Epoch 18 / 20) train acc: 0.702000; val_acc: 0.531000
(Iteration 8821 / 9800) loss: 0.678524
(Iteration 8831 / 9800) loss: 0.804313
(Iteration 8841 / 9800) loss: 0.803434
(Iteration 8851 / 9800) loss: 0.754743
(Iteration 8861 / 9800) loss: 0.743720
(Iteration 8871 / 9800) loss: 0.890108
(Iteration 8881 / 9800) loss: 0.805879
(Iteration 8891 / 9800) loss: 0.952859
(Iteration 8901 / 9800) loss: 0.967723
(Iteration 8911 / 9800) loss: 0.932334
(Iteration 8921 / 9800) loss: 0.867855
(Iteration 8931 / 9800) loss: 0.916111
(Iteration 8941 / 9800) loss: 0.964982
(Iteration 8951 / 9800) loss: 0.996291
(Iteration 8961 / 9800) loss: 0.814213
(Iteration 8971 / 9800) loss: 1.137798
(Iteration 8981 / 9800) loss: 0.892646
(Iteration 8991 / 9800) loss: 0.842733
(Iteration 9001 / 9800) loss: 0.879668
(Iteration 9011 / 9800) loss: 0.805979
(Iteration 9021 / 9800) loss: 1.034354
(Iteration 9031 / 9800) loss: 0.897894
(Iteration 9041 / 9800) loss: 0.844538
(Iteration 9051 / 9800) loss: 0.806714
(Iteration 9061 / 9800) loss: 0.895295
(Iteration 9071 / 9800) loss: 0.948517
(Iteration 9081 / 9800) loss: 0.871645
(Iteration 9091 / 9800) loss: 0.831883
(Iteration 9101 / 9800) loss: 0.845667
(Iteration 9111 / 9800) loss: 0.743608
(Iteration 9121 / 9800) loss: 0.832206
(Iteration 9131 / 9800) loss: 0.845785
(Iteration 9141 / 9800) loss: 0.905356
(Iteration 9151 / 9800) loss: 0.957419
(Iteration 9161 / 9800) loss: 0.719037
(Iteration 9171 / 9800) loss: 0.885659
(Iteration 9181 / 9800) loss: 0.790388
(Iteration 9191 / 9800) loss: 0.855506
```

```
(Iteration 9201 / 9800) loss: 0.729800
(Iteration 9211 / 9800) loss: 0.732475
(Iteration 9221 / 9800) loss: 0.861116
(Iteration 9231 / 9800) loss: 0.960030
(Iteration 9241 / 9800) loss: 0.859740
(Iteration 9251 / 9800) loss: 1.006888
(Iteration 9261 / 9800) loss: 0.902544
(Iteration 9271 / 9800) loss: 0.884126
(Iteration 9281 / 9800) loss: 0.744725
(Iteration 9291 / 9800) loss: 0.718549
(Iteration 9301 / 9800) loss: 1.122717
(Epoch 19 / 20) train acc: 0.720000; val_acc: 0.544000
(Iteration 9311 / 9800) loss: 0.927243
(Iteration 9321 / 9800) loss: 1.024728
(Iteration 9331 / 9800) loss: 0.724476
(Iteration 9341 / 9800) loss: 0.920898
(Iteration 9351 / 9800) loss: 0.878088
(Iteration 9361 / 9800) loss: 0.851232
(Iteration 9371 / 9800) loss: 1.136103
(Iteration 9381 / 9800) loss: 0.755134
(Iteration 9391 / 9800) loss: 0.748980
(Iteration 9401 / 9800) loss: 0.862709
(Iteration 9411 / 9800) loss: 0.763628
(Iteration 9421 / 9800) loss: 0.805095
(Iteration 9431 / 9800) loss: 1.033096
(Iteration 9441 / 9800) loss: 0.929374
(Iteration 9451 / 9800) loss: 0.868767
(Iteration 9461 / 9800) loss: 0.897477
(Iteration 9471 / 9800) loss: 0.686802
(Iteration 9481 / 9800) loss: 0.762822
(Iteration 9491 / 9800) loss: 1.040477
(Iteration 9501 / 9800) loss: 0.754824
(Iteration 9511 / 9800) loss: 0.835452
(Iteration 9521 / 9800) loss: 0.794501
(Iteration 9531 / 9800) loss: 0.974651
(Iteration 9541 / 9800) loss: 0.820483
(Iteration 9551 / 9800) loss: 0.543222
(Iteration 9561 / 9800) loss: 0.828670
(Iteration 9571 / 9800) loss: 0.692911
(Iteration 9581 / 9800) loss: 1.051020
(Iteration 9591 / 9800) loss: 1.005018
(Iteration 9601 / 9800) loss: 0.636251
(Iteration 9611 / 9800) loss: 1.119445
(Iteration 9621 / 9800) loss: 1.078424
(Iteration 9631 / 9800) loss: 1.000789
(Iteration 9641 / 9800) loss: 0.693634
(Iteration 9651 / 9800) loss: 0.895479
(Iteration 9661 / 9800) loss: 0.816582
```

```
(Iteration 9671 / 9800) loss: 0.804479
(Iteration 9681 / 9800) loss: 0.793157
(Iteration 9691 / 9800) loss: 0.821866
(Iteration 9701 / 9800) loss: 0.728754
(Iteration 9711 / 9800) loss: 0.840363
(Iteration 9721 / 9800) loss: 0.879655
(Iteration 9731 / 9800) loss: 0.831423
(Iteration 9741 / 9800) loss: 0.836845
(Iteration 9751 / 9800) loss: 0.845199
(Iteration 9761 / 9800) loss: 0.884415
(Iteration 9771 / 9800) loss: 0.836210
(Iteration 9781 / 9800) loss: 0.748419
(Iteration 9791 / 9800) loss: 0.864301
(Epoch 20 / 20) train acc: 0.717000; val_acc: 0.551000
```



4 Test Your Model!

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

[15]:
```
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:  0.518
Test set accuracy:  0.515
```

BatchNormalization

April 30, 2023

[8]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs231n/assignments/assignment2/'
# FOLDERNAME = None
# 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

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, proposed by [1] in 2015.

To understand the goal of batch normalization, it is important to first recognize that machine learning methods tend to perform better with input data consisting 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, they propose to insert into the network layers that normalize batches. At training time, such a 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.

[9]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

%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(f"  means: {x.mean(axis=axis)}")
    print(f"  stds:  {x.std(axis=axis)}\n")
```

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

[10]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Batch Normalization: Forward Pass

In the file `cs231n/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!

[11]:
```
# 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: [5.32907052e-17 7.04991621e-17 1.85962357e-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]
```

[12]:
```python
# 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]
```

3 Batch Normalization: Backward Pass

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.

[13]:
```
# 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:  1.7029261167605239e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

4 Batch Normalization: Alternative Backward Pass

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/hard 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.

[14]:
```
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:  2.39488637878453e-11
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.36x
```

5 Fully Connected Networks with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `cs231n/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 `cs231n/layer_utils.py`.

[15]:
```
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: 3.35e-06
W3 relative error: 3.92e-10
b1 relative error: 2.66e-07
b2 relative error: 4.44e-08
b3 relative error: 8.26e-11
beta1 relative error: 7.85e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 7.47e-09
gamma2 relative error: 3.35e-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: 1.94e-08
b2 relative error: 8.22e-07
b3 relative error: 1.73e-10
beta1 relative error: 6.32e-09
beta2 relative error: 3.48e-09
gamma1 relative error: 5.94e-09
gamma2 relative error: 4.67e-09
```

6 Batch Normalization 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.

[16]:
```
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()
```

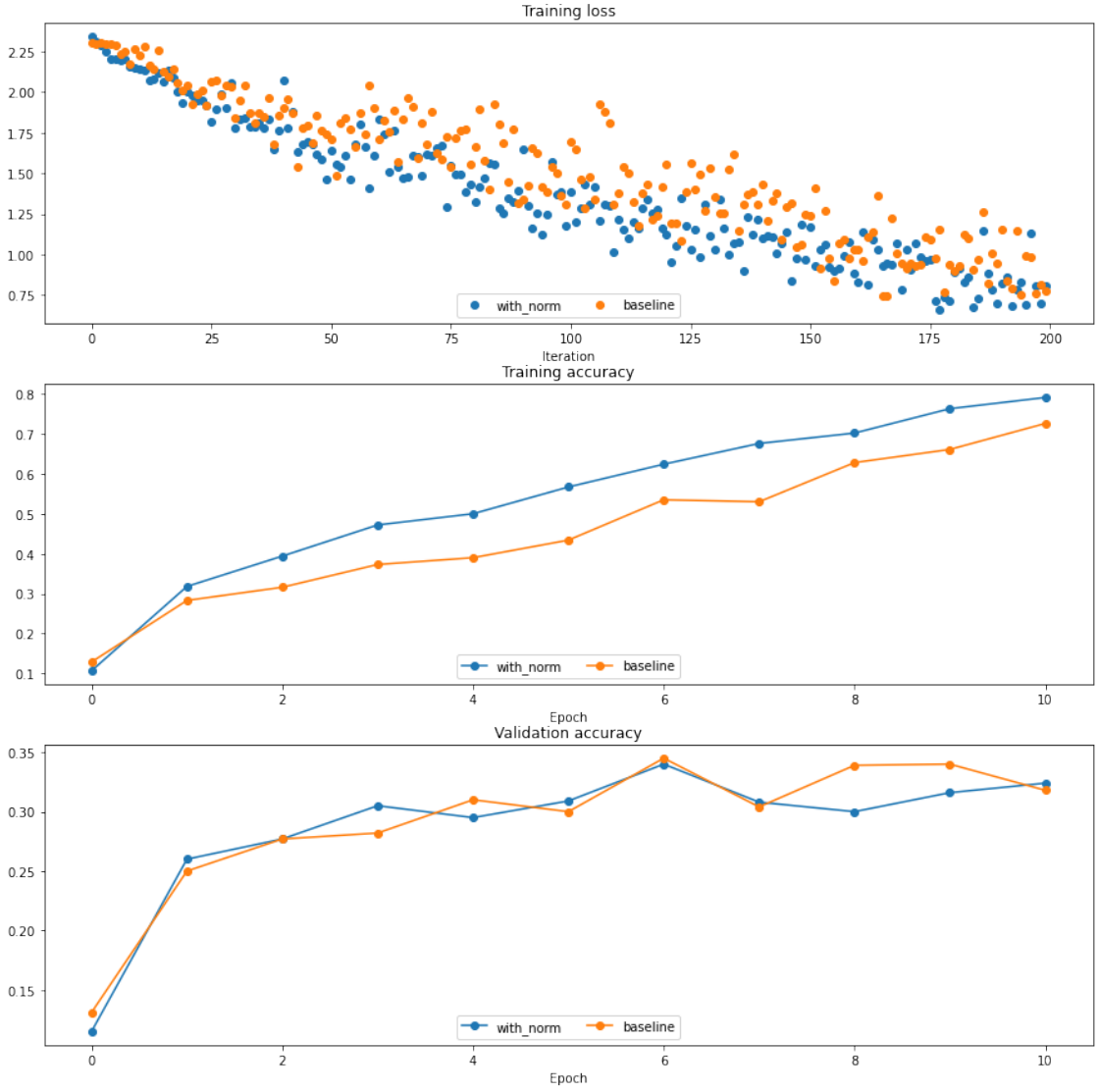
```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.318000; val_acc: 0.260000
(Iteration 21 / 200) loss: 2.000091
(Epoch 2 / 10) train acc: 0.394000; val_acc: 0.277000
(Iteration 41 / 200) loss: 2.071344
(Epoch 3 / 10) train acc: 0.472000; val_acc: 0.305000
(Iteration 61 / 200) loss: 1.835723
(Epoch 4 / 10) train acc: 0.500000; val_acc: 0.295000
(Iteration 81 / 200) loss: 1.321223
(Epoch 5 / 10) train acc: 0.567000; val_acc: 0.309000
(Iteration 101 / 200) loss: 1.388430
(Epoch 6 / 10) train acc: 0.624000; val_acc: 0.340000
(Iteration 121 / 200) loss: 1.126643
(Epoch 7 / 10) train acc: 0.676000; val_acc: 0.308000
(Iteration 141 / 200) loss: 1.101064
(Epoch 8 / 10) train acc: 0.702000; val_acc: 0.300000
(Iteration 161 / 200) loss: 0.830944
(Epoch 9 / 10) train acc: 0.763000; val_acc: 0.316000
(Iteration 181 / 200) loss: 0.893360
(Epoch 10 / 10) train acc: 0.791000; 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: 1.662209
(Epoch 5 / 10) train acc: 0.434000; val_acc: 0.300000
(Iteration 101 / 200) loss: 1.696059
(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: 1.033931
(Epoch 9 / 10) train acc: 0.661000; val_acc: 0.340000
(Iteration 181 / 200) loss: 0.901034
(Epoch 10 / 10) train acc: 0.726000; val_acc: 0.318000
```

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.

[17]:
```
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()
```



7 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 eight-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.

[18]:
```
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
```

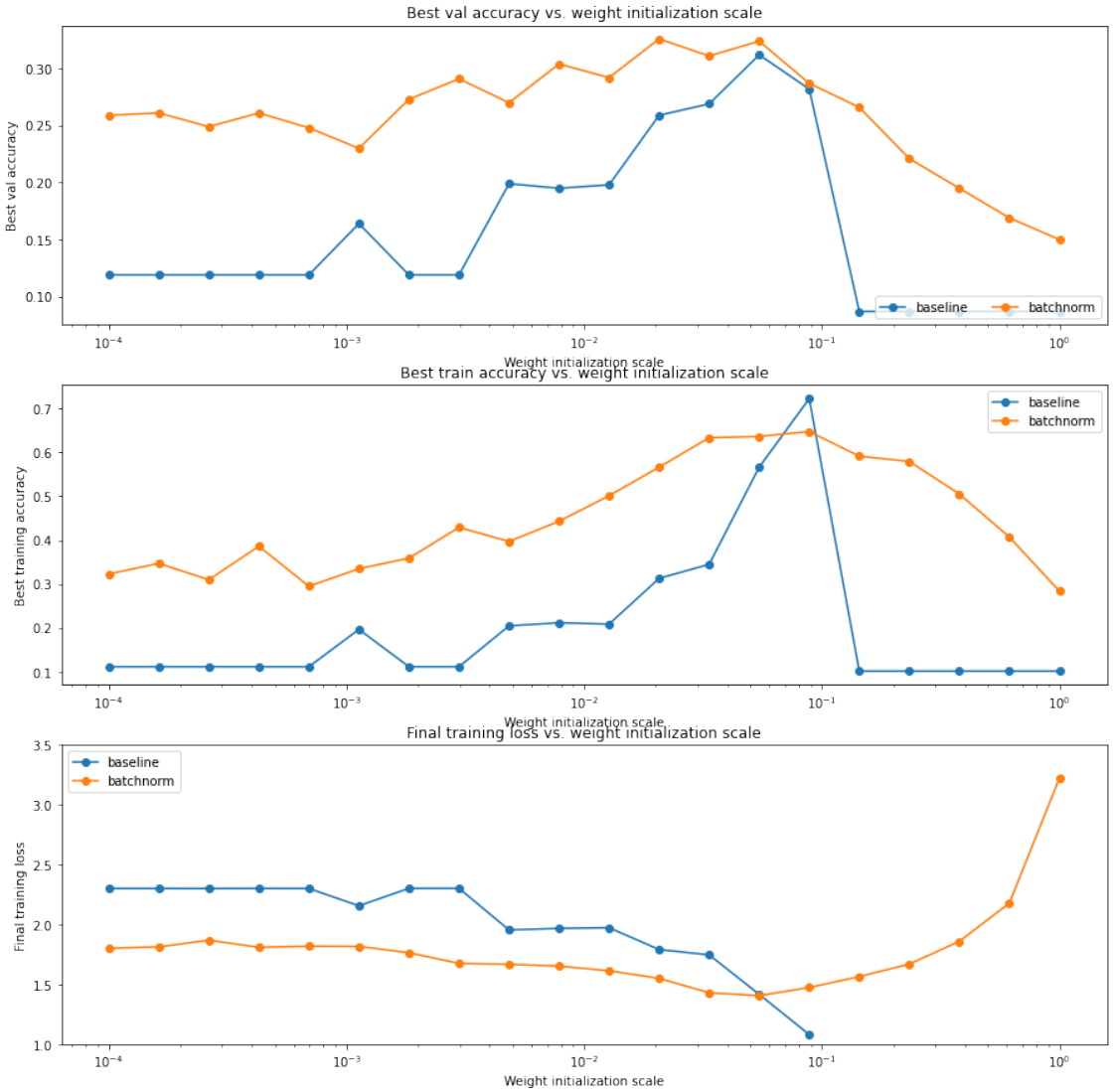
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 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
```

```
/Users/paul-emile/Documents/CS 231N Homeworks/assignment2/cs231n/layers.py:147:
RuntimeWarning: overflow encountered in exp
  loss = -np.sum(np.log(np.exp(correct_class_scores)/np.sum(np.exp(scores),
axis=1)))
/Users/paul-emile/Documents/CS 231N Homeworks/assignment2/cs231n/layers.py:147:
RuntimeWarning: invalid value encountered in divide
  loss = -np.sum(np.log(np.exp(correct_class_scores)/np.sum(np.exp(scores),
axis=1)))
/Users/paul-emile/Documents/CS 231N Homeworks/assignment2/cs231n/layers.py:147:
RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(np.exp(correct_class_scores)/np.sum(np.exp(scores),
axis=1)))
/Users/paul-emile/Documents/CS 231N Homeworks/assignment2/cs231n/layers.py:150:
RuntimeWarning: overflow encountered in exp
  dx = np.exp(scores)/np.sum(np.exp(scores), axis=1).reshape(num_train,1)
/Users/paul-emile/Documents/CS 231N Homeworks/assignment2/cs231n/layers.py:150:
RuntimeWarning: invalid value encountered in divide
  dx = np.exp(scores)/np.sum(np.exp(scores), axis=1).reshape(num_train,1)
```

```
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

[19]:
```
# 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()
```



7.1 Inline Question 1:

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

7.2 Answer:

The validation and training accuracy, as well as the loss of the model can be sensitive to the weight initialization scale if it does not include batch normalization. We can see it on the graphs and in the RunTimeWarnings above: numerical errors can occur due to values that can be too big or too small. With batch normalization, the model becomes robust and is not as much affacted by the weight initialization scale.

Without batch normalization, if the initial weights are too large or too small, the activation values

can explode or vanish, causing numerical instability and making the optimization difficult. This can lead to slow convergence and poor generalization.

With batch normalization, the model is less sensitive to the choice of weight initialization scale. Batch normalization applies a normalization transformation to the inputs of each layer by adjusting their mean and standard deviation. This helps to keep the activation values within a reasonable range, thus avoiding instability issues. As a result, the model can be trained with larger learning rates and train faster.

8 Batch Normalization and Batch Size

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

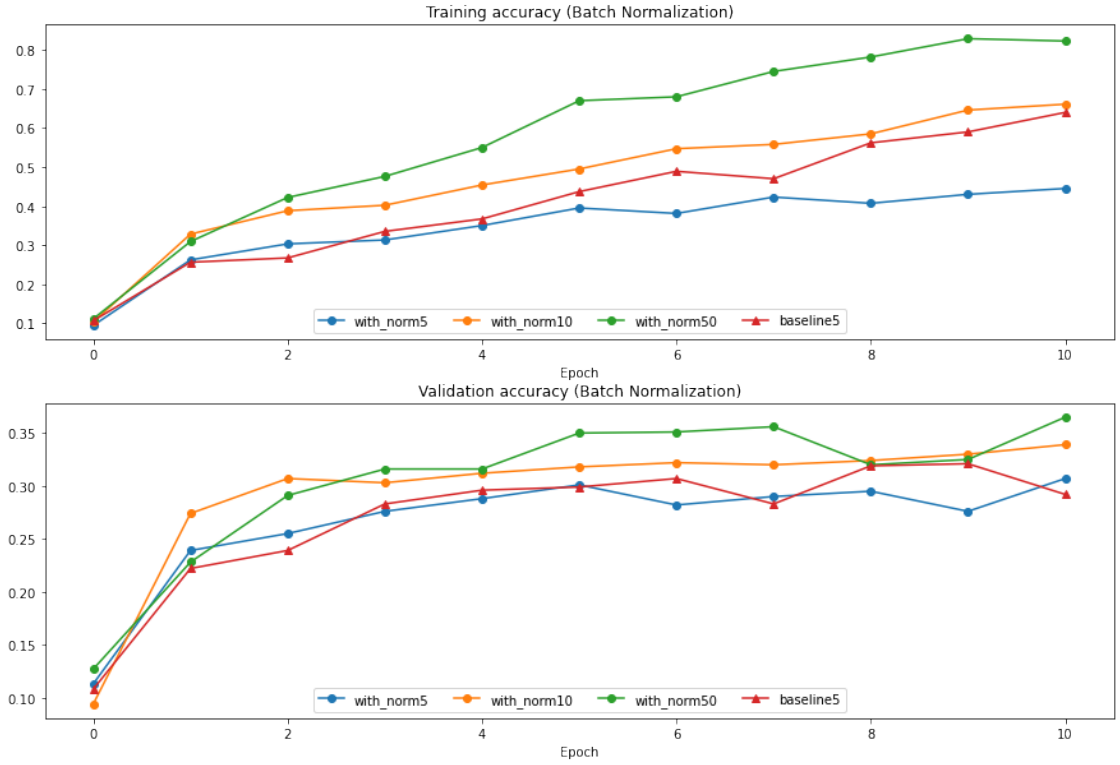
The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

[20]:
```
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
```

[21]:
```
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()
```



8.1 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?

8.2 Answer:

We can see on the graphs above that batch normalization with a batch size of 5 has a lower training accuracy than with a batch size of 50, but the validation accuracy is higher with a batch size of 5 than 50. The batch ize of 10 has a behavior that is somewhat in between the other two. We thus observe that increasing the batch size may lead to overfitting.

Moreover, increasing the batch size will increase the training time as the computation time for the batch normalization will be bigger.

Finally, increasing the batch size will lead to a more accurate normalization as the bigger the sample, the better the approximation of the mean and the variance.

Therefore, there is a tradeoff between normalization accuracy, training time and overfitting when it comes to choosing a batch size.

9 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.

9.1 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.

9.2 Answer:

1. Layer normalization (each row is treated as a feature)
2. Batch normalization (helps to reduce the variance)
3. Batch normalization (helps center the data)
4. None

10 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 `cs231n/layers.py`, implement the forward pass for layer normalization in the function `layernorm_forward`.

Run the cell below to check your results. * In `cs231n/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `cs231n/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.

[22]:
```
# 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: [ 4.81096644e-16 -7.40148683e-17  2.22044605e-16 -5.92118946e-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]
```

[23]:
```
# 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))
```

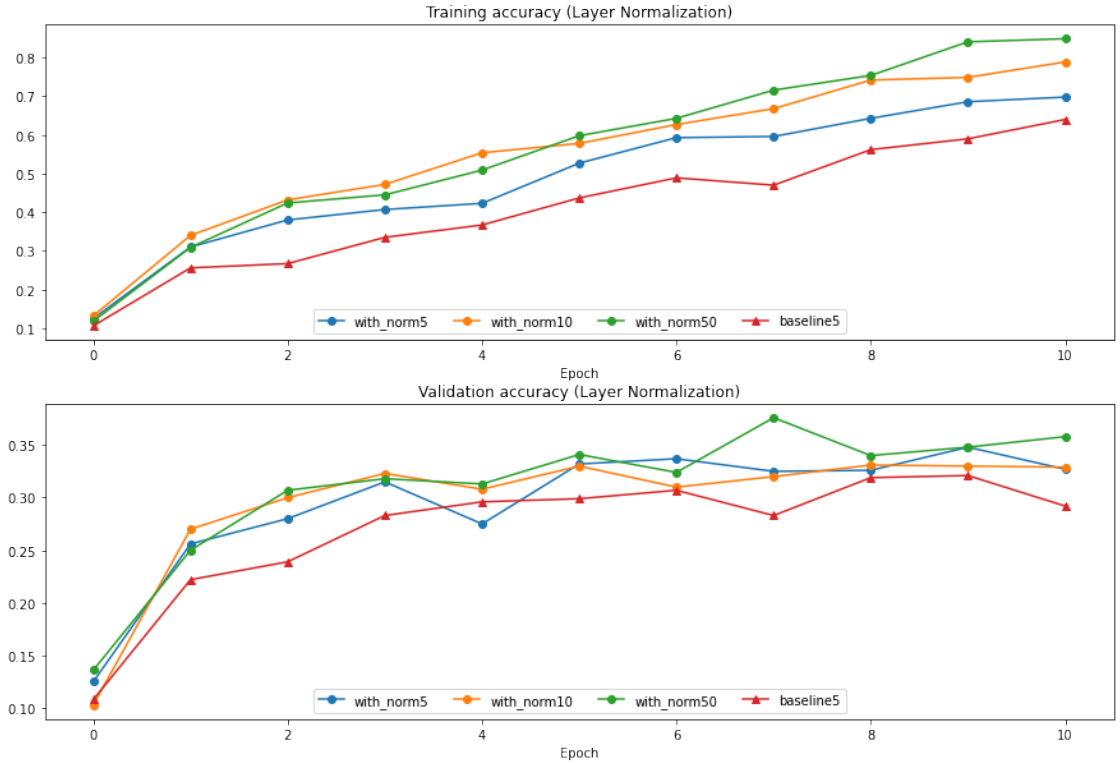
```
dx error:  1.4336158494902849e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12
```

11 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!

[24]:
```
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
```



11.1 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

11.2 Answer:

1. If the very deep network has regular connections, layer normaliation should work well.
2. If we have a very small dimension of features, layer normalization is likely not to work well. Because of this small dimension of features, the means and standard deviations computed with layer normalization are likely to not be representative of the true underlying feature distribution.
3. If we have a high regularization term, it can restrict the range of values that the parameters can take, and layer normilaztion may not capture the true underlying feature distribution.

Dropout

April 30, 2023

[9]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs231n/assignments/assignment2/'
# FOLDERNAME = None
# 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

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

[10]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
```

```
from cs231n.gradient_check import eval_numerical_gradient,␣
  ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

%load_ext autoreload
%autoreload 2

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

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

[11]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Dropout: Forward Pass

In the file `cs231n/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.

[12]:
```
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 `cs231n/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

[13]:
```
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:

We divide by `p` during training time to get the same mean value of inputs to each layer as in test time. If we do not do this, the outputs of each layer will have values that are too small on average (mean) during test time compared to the case where the network was trained with the division by `p`. This will lead to a smaller accuracy.

4 Fully Connected Networks with Dropout

In the file `cs231n/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_keep_ratio` 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.

[14]:
```
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_keep_ratio in [1, 0.75, 0.5]:
    print('Running check with dropout = ', dropout_keep_ratio)
    model = FullyConnectedNet(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        weight_scale=5e-2,
        dtype=np.float64,
        dropout_keep_ratio=dropout_keep_ratio,
        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_keep_ratio=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: 7.70e-08
W2 relative error: 1.71e-05
W3 relative error: 2.95e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 6.60e-11

Running check with dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.85e-07
W2 relative error: 2.15e-06
W3 relative error: 4.56e-08
b1 relative error: 1.16e-08
b2 relative error: 1.82e-09
b3 relative error: 1.40e-10

Running check with dropout =  0.5
Initial loss:  2.30427592207859
W1 relative error: 3.11e-07
W2 relative error: 5.55e-08
W3 relative error: 6.47e-08
b1 relative error: 2.58e-08
b2 relative error: 1.91e-09
b3 relative error: 9.31e-11
```

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.

[36]:
```
# 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 = {}
```

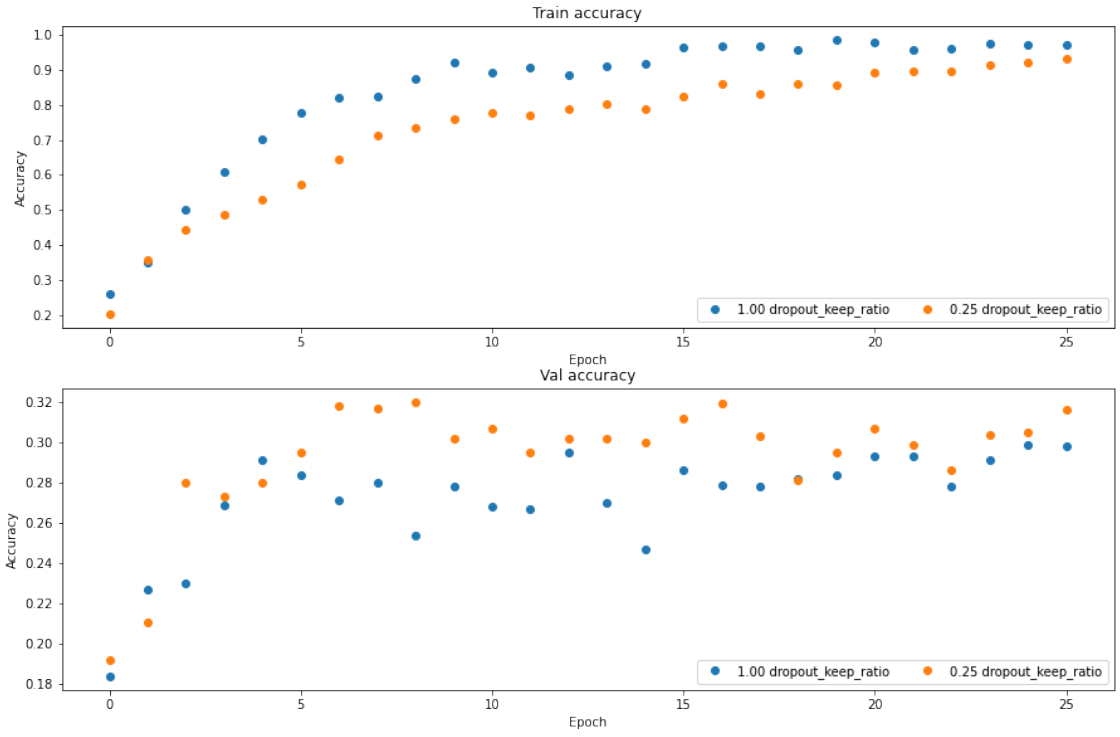
```
dropout_choices = [0.9999999999, 0.25]
for dropout_keep_ratio in dropout_choices:
    model = FullyConnectedNet(
        [500],
        dropout_keep_ratio=dropout_keep_ratio,
    )
    print(dropout_keep_ratio)

    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_keep_ratio] = solver
    print()
```

```
0.9999999999
(Iteration 1 / 125) loss: 7.856644
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.350000; val_acc: 0.227000
(Epoch 2 / 25) train acc: 0.500000; val_acc: 0.230000
(Epoch 3 / 25) train acc: 0.610000; val_acc: 0.269000
(Epoch 4 / 25) train acc: 0.704000; val_acc: 0.291000
(Epoch 5 / 25) train acc: 0.778000; val_acc: 0.284000
(Epoch 6 / 25) train acc: 0.820000; val_acc: 0.271000
(Epoch 7 / 25) train acc: 0.824000; val_acc: 0.280000
(Epoch 8 / 25) train acc: 0.876000; val_acc: 0.254000
(Epoch 9 / 25) train acc: 0.922000; val_acc: 0.278000
(Epoch 10 / 25) train acc: 0.894000; val_acc: 0.268000
(Epoch 11 / 25) train acc: 0.906000; val_acc: 0.267000
(Epoch 12 / 25) train acc: 0.886000; val_acc: 0.295000
(Epoch 13 / 25) train acc: 0.912000; val_acc: 0.270000
(Epoch 14 / 25) train acc: 0.918000; val_acc: 0.247000
(Epoch 15 / 25) train acc: 0.966000; val_acc: 0.286000
(Epoch 16 / 25) train acc: 0.970000; val_acc: 0.279000
(Epoch 17 / 25) train acc: 0.968000; val_acc: 0.278000
(Epoch 18 / 25) train acc: 0.958000; val_acc: 0.282000
(Epoch 19 / 25) train acc: 0.986000; val_acc: 0.284000
(Epoch 20 / 25) train acc: 0.980000; val_acc: 0.293000
(Iteration 101 / 125) loss: 0.346716
(Epoch 21 / 25) train acc: 0.958000; val_acc: 0.293000
```

```
(Epoch 22 / 25) train acc: 0.960000; val_acc: 0.278000
(Epoch 23 / 25) train acc: 0.974000; val_acc: 0.291000
(Epoch 24 / 25) train acc: 0.972000; val_acc: 0.299000
(Epoch 25 / 25) train acc: 0.972000; val_acc: 0.298000

0.25
(Iteration 1 / 125) loss: 17.045779
(Epoch 0 / 25) train acc: 0.202000; val_acc: 0.192000
(Epoch 1 / 25) train acc: 0.358000; val_acc: 0.211000
(Epoch 2 / 25) train acc: 0.444000; val_acc: 0.280000
(Epoch 3 / 25) train acc: 0.488000; val_acc: 0.273000
(Epoch 4 / 25) train acc: 0.530000; val_acc: 0.280000
(Epoch 5 / 25) train acc: 0.574000; val_acc: 0.295000
(Epoch 6 / 25) train acc: 0.644000; val_acc: 0.318000
(Epoch 7 / 25) train acc: 0.714000; val_acc: 0.317000
(Epoch 8 / 25) train acc: 0.736000; val_acc: 0.320000
(Epoch 9 / 25) train acc: 0.760000; val_acc: 0.302000
(Epoch 10 / 25) train acc: 0.776000; val_acc: 0.307000
(Epoch 11 / 25) train acc: 0.770000; val_acc: 0.295000
(Epoch 12 / 25) train acc: 0.790000; val_acc: 0.302000
(Epoch 13 / 25) train acc: 0.804000; val_acc: 0.302000
(Epoch 14 / 25) train acc: 0.788000; val_acc: 0.300000
(Epoch 15 / 25) train acc: 0.824000; val_acc: 0.312000
(Epoch 16 / 25) train acc: 0.860000; val_acc: 0.319000
(Epoch 17 / 25) train acc: 0.830000; val_acc: 0.303000
(Epoch 18 / 25) train acc: 0.860000; val_acc: 0.281000
(Epoch 19 / 25) train acc: 0.856000; val_acc: 0.295000
(Epoch 20 / 25) train acc: 0.894000; val_acc: 0.307000
(Iteration 101 / 125) loss: 5.297325
(Epoch 21 / 25) train acc: 0.898000; val_acc: 0.299000
(Epoch 22 / 25) train acc: 0.896000; val_acc: 0.286000
(Epoch 23 / 25) train acc: 0.916000; val_acc: 0.304000
(Epoch 24 / 25) train acc: 0.922000; val_acc: 0.305000
(Epoch 25 / 25) train acc: 0.934000; val_acc: 0.316000
```

[37]:
```
# Plot train and validation accuracies of the two models.
train_accs = []
val_accs = []
for dropout_keep_ratio in dropout_choices:
    solver = solvers[dropout_keep_ratio]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
```

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

plt.subplot(3, 1, 2)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
        solvers[dropout_keep_ratio].val_acc_history, 'o', label='%.2f␣
 ↪dropout_keep_ratio' % dropout_keep_ratio)
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:

Using dropout as a regularizing effect. The network has a lower training accuracy and a higher validation accuracy. This demonstrates that the network is able to generalize more and we avoid overfitting effects.

ConvolutionalNetworks

April 30, 2023

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

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs231n/assignments/assignment2/'
# FOLDERNAME = None
# 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Convolutional Networks

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

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

[3]:
```
# Setup cell.
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
```

```
from cs231n.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

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

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

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

[4]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Convolution: Naive Forward Pass

The core of a convolutional network is the convolution operation. In the file `cs231n/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

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

You can test your implementation by running the following:

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

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

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

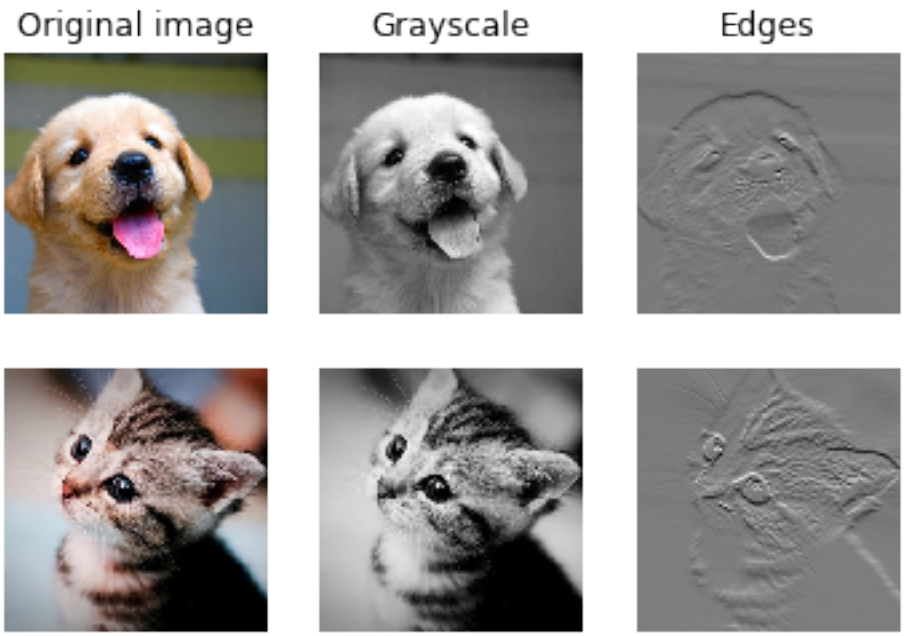
```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

2.1 Aside: Image Processing via Convolutions

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

[5]:
```
from imageio import imread
from PIL import Image

kitten = imread('cs231n/notebook_images/kitten.jpg')
puppy = imread('cs231n/notebook_images/puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_puppy = np.array(Image.fromarray(puppy).resize((img_size, img_size)))
resized_kitten = np.array(Image.fromarray(kitten_cropped).resize((img_size,␣
 ↪img_size)))
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = resized_puppy.transpose((2, 0, 1))
x[1, :, :, :] = resized_kitten.transpose((2, 0, 1))
```

```
# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

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

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

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

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

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



3 Convolution: Naive Backward Pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

[6]:
```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

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

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

# Your errors should be around e-8 or less.
```

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

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

4 Max-Pooling: Naive Forward Pass

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

[7]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

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

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

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

5 Max-Pooling: Naive Backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. You don't need to worry about

computational efficiency.

Check your implementation with numeric gradient checking by running the following:

[8]:
```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

# Your error should be on the order of e-12
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

6 Fast Layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py`.

6.0.1 Execute the below cell, save the notebook, and restart the runtime

The fast convolution implementation depends on a Cython extension; to compile it, run the cell below. Next, save the Colab notebook (`File > Save`) and **restart the runtime** (`Runtime > Restart runtime`). You can then re-execute the preceeding cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

[9]:
```
# # Remember to restart the runtime after executing this cell!
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/
# !python setup.py build_ext --inplace
# %cd /content/drive/My\ Drive/$FOLDERNAME/
```

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

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

You can compare the performance of the naive and fast versions of these layers by running the following:

[10]:
```
# Rel errors should be around e-9 or less.
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

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

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

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

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

```
Testing conv_forward_fast:
Naive: 1.714587s
Fast: 0.005056s
Speedup: 339.109728x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 2.960095s
Fast: 0.010404s
```

```
Speedup: 284.518620x
dx difference:  1.949764775345631e-11
dw difference:  4.428294734673248e-13
db difference:  3.481354613192702e-14
```

[12]:
```python
# Relative errors should be close to 0.0.
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

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

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

```
Testing pool_forward_fast:
Naive: 0.139911s
fast: 0.001317s
speedup: 106.232621x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.197193s
fast: 0.005761s
speedup: 34.228067x
dx difference:  0.0
```

7 Convolutional "Sandwich" Layers

In the previous assignment, we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check their usage.

[13]:
```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

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

```
Testing conv_relu_pool
dx error:  9.591132621921372e-09
dw error:  5.802401370096438e-09
db error:  3.57960501324485e-10
```

[14]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

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

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

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

```
Testing conv_relu:
dx error:  1.5218619980349303e-09
dw error:  3.3716088557723477e-10
db error:  4.8422803898140394e-11
```

8 Three-Layer Convolutional Network

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

Open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

8.1 Sanity Check Loss

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

[16]:
```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

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

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

```
Initial loss (no regularization):  2.302585412176879
Initial loss (with regularization):  2.508542482169898
```

8.2 Gradient Check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

[17]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(
    num_filters=3,
    filter_size=3,
    input_dim=input_dim,
    hidden_dim=7,
    dtype=np.float64
)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],␣
 ↪verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
 ↪grads[param_name])))
```

```
W1 max relative error: 4.076677e-04
W2 max relative error: 1.493310e-02
W3 max relative error: 3.422460e-04
b1 max relative error: 3.397321e-06
b2 max relative error: 2.517459e-03
b3 max relative error: 1.007692e-09
```

8.3 Overfit Small Data

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

[18]:
```
np.random.seed(231)

num_train = 100
```

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

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(
    model,
    small_data,
    num_epochs=15,
    batch_size=50,
    update_rule='adam',
    optim_config={'learning_rate': 1e-3,},
    verbose=True,
    print_every=1
)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
```

```
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

[19]:
```
# Print final training accuracy.
print(
    "Small data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
```

```
Small data training accuracy: 0.82
```

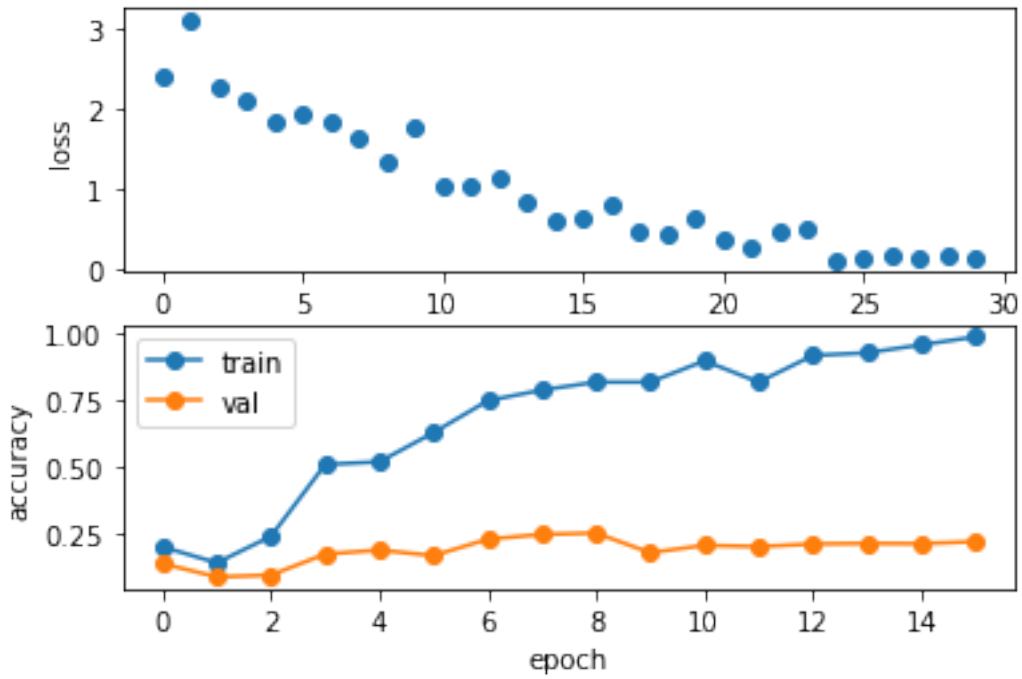
[20]:
```
# Print final validation accuracy.
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

```
Small data validation accuracy: 0.252
```

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

[21]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

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

```
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



8.4 Train the Network

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

[22]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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

```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
```

```
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
```

```
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

[23]:
```
# Print final training accuracy.
print(
    "Full data training accuracy:",
    solver.check_accuracy(data['X_train'], data['y_train'])
)
```

```
Full data training accuracy: 0.4761836734693878
```

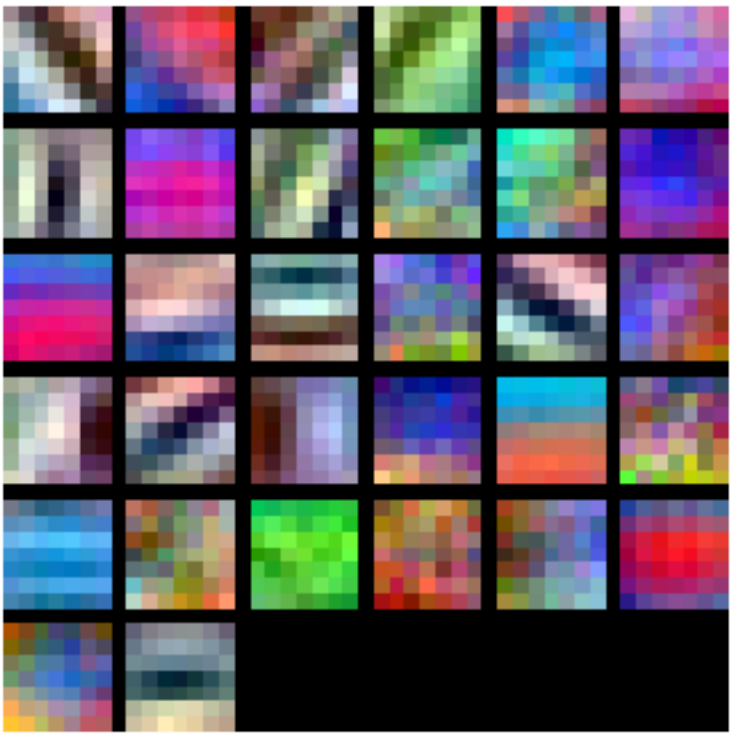
[24]:
```
# Print final validation accuracy.
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
```

```
Full data validation accuracy: 0.499
```

8.5 Visualize Filters

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

[25]:
```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



9 Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully connected networks. As proposed in the original paper (link in `BatchNormalization.ipynb`), batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

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

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

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

10 Spatial Batch Normalization: Forward Pass

In the file `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

[97]:
```
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]
```

[108]:
```
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]
```

11 Spatial Batch Normalization: Backward Pass

In the file `cs231n/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:

[110]:
```
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:  2.786648197756335e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

12 Spatial Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [2] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [3] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups and a per-group per-datapoint normalization instead.

Visual comparison of the normalization techniques discussed so far (image edited from [3])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional computer vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [4] – after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization.

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

[3] Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.

13 Spatial Group Normalization: Forward Pass

In the file `cs231n/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

[118]:
```
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, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  shape: ', x.shape)
print('  means: ', x_g.mean(axis=1))
print('  stds: ', x_g.std(axis=1))

# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  shape: ', out.shape)
print('  means: ', out_g.mean(axis=1))
print('  stds: ', out_g.std(axis=1))
```

```
Before spatial group normalization:
  shape:  (2, 6, 4, 5)
  means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  shape:  (2, 6, 4, 5)
  means:  [-2.14643118e-16  5.25505565e-16  2.65528340e-16 -3.38618023e-16]
  stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

14 Spatial Group Normalization: Backward Pass

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[24]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
```

```
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

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

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)

# You should expect errors of magnitudes between 1e-12 and 1e-07.
print(da_num.shape)
print(dgamma.shape)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
(1, 6, 1, 1)
(1, 6, 1, 1)
dx error:  7.413109384854475e-08
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12
```

PyTorch

April 30, 2023

[6]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs231n/assignments/assignment2/'
# FOLDERNAME = None
# 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Introduction to PyTorch

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.

1.1 Why do we use deep learning frameworks?

- Our code will now run on GPUs! This will allow our models to train much faster. When using a framework like PyTorch 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).
- In 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! PyTorch is an excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- Finally, we want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.2 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.

1.3 How do I learn PyTorch?

One of our former instructors, Justin Johnson, 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.

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 GPU

You can manually switch to a GPU device on Colab by clicking `Runtime -> Change runtime type` and selecting `GPU` under `Hardware Accelerator`. You should do this before running the following cells to import packages, since the kernel gets restarted upon switching runtimes.

[56]:
```
import torch
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

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: cpu
```

4 Part I. Preparation

Now, let's 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.

[8]:
```
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('./cs231n/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('./cs231n/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('./cs231n/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
```

5 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.

5.0.1 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).

[9]:
```
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]])
```

5.0.2 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.

[10]:
```python
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])
```

5.0.3 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!

[11]:
```
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? --> (H*W*channel_2, C)
    - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can␣
↪you
      figure out what the shape should be? --> (C,)

  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.conv2d(input=x, weight=conv_w1, bias=conv_b1, padding=2)
  scores = F.relu(scores)
  scores = F.conv2d(input=scores, weight=conv_w2, bias=conv_b2, padding=1)
  scores = F.relu(scores)
  scores = flatten(scores)
  scores =  scores @ fc_w + 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).

[12]:
```
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])
```

5.0.4 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

[13]:
```
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))
```

[13]:
```
tensor([[ 0.4336,  0.7713, -0.0912, -0.9308,  0.2026],
        [-0.7243,  0.7703,  0.0386, -0.7122, -0.0360],
        [ 1.0792, -0.0926, -0.1740,  1.3198, -1.2064]], requires_grad=True)
```

5.0.5 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.

[14]:
```
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))
```

5.0.6 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.

[15]:
```
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()
```

5.0.7 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.

[16]:
```
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 = 3.9249
Checking accuracy on the val set
Got 132 / 1000 correct (13.20%)

Iteration 100, loss = 2.1340
Checking accuracy on the val set
Got 368 / 1000 correct (36.80%)

Iteration 200, loss = 1.7662
Checking accuracy on the val set
Got 375 / 1000 correct (37.50%)

Iteration 300, loss = 1.9311
Checking accuracy on the val set
Got 397 / 1000 correct (39.70%)

Iteration 400, loss = 1.7280
Checking accuracy on the val set
Got 369 / 1000 correct (36.90%)
```

```
Iteration 500, loss = 2.0138
Checking accuracy on the val set
Got 422 / 1000 correct (42.20%)

Iteration 600, loss = 1.5840
Checking accuracy on the val set
Got 431 / 1000 correct (43.10%)

Iteration 700, loss = 1.7156
Checking accuracy on the val set
Got 446 / 1000 correct (44.60%)
```

5.0.8 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.

[17]:
```
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)*****

KH1, KW1 = 5, 5
```

```
KH2, KW2 = 3, 3

conv_w1 = random_weight((channel_1, 3, KH1, KW1))
conv_b1 = zero_weight((channel_1,))
conv_w2 = random_weight((channel_2, channel_1, KH2, KW2))
conv_b2 = zero_weight((channel_2,))
fc_w = random_weight((32*32*channel_2, 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 = 4.9783
Checking accuracy on the val set
Got 117 / 1000 correct (11.70%)

Iteration 100, loss = 1.9622
Checking accuracy on the val set
Got 339 / 1000 correct (33.90%)

Iteration 200, loss = 1.6834
Checking accuracy on the val set
Got 376 / 1000 correct (37.60%)

Iteration 300, loss = 1.8261
Checking accuracy on the val set
Got 410 / 1000 correct (41.00%)

Iteration 400, loss = 1.4393
Checking accuracy on the val set
Got 415 / 1000 correct (41.50%)

Iteration 500, loss = 1.7728
Checking accuracy on the val set
Got 440 / 1000 correct (44.00%)

Iteration 600, loss = 1.3948
Checking accuracy on the val set
Got 459 / 1000 correct (45.90%)

Iteration 700, loss = 1.3940
Checking accuracy on the val set
```

```
Got 478 / 1000 correct (47.80%)
```

6 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.

6.0.1 Module API: Two-Layer Network

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

[18]:
```python
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])
```

6.0.2 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.

[19]:
```
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_channel, channel_1, kernel_size=5, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)

        self.activation1 = F.relu

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

```
        self.activation2 = F.relu

        self.fc1 = nn.Linear(32*32*channel_2, num_classes)
        nn.init.kaiming_normal_(self.fc1.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.conv1(x)
        scores = self.activation1(scores)
        scores = self.conv2(scores)
        scores = self.activation2(scores)
        scores = flatten(scores)
        scores = self.fc1(scores)

        # *****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])
```

6.0.3 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.

[20]:
```
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))
```

6.0.4 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.

[21]:
```
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()
```

6.0.5 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.

[22]:
```
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 = 3.3894
Checking accuracy on validation set
Got 144 / 1000 correct (14.40)

Iteration 100, loss = 2.4234
Checking accuracy on validation set
Got 353 / 1000 correct (35.30)
```

```
Iteration 200, loss = 1.8274
Checking accuracy on validation set
Got 382 / 1000 correct (38.20)

Iteration 300, loss = 1.6858
Checking accuracy on validation set
Got 335 / 1000 correct (33.50)

Iteration 400, loss = 2.5971
Checking accuracy on validation set
Got 329 / 1000 correct (32.90)

Iteration 500, loss = 1.7974
Checking accuracy on validation set
Got 419 / 1000 correct (41.90)

Iteration 600, loss = 1.4780
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Iteration 700, loss = 1.7016
Checking accuracy on validation set
Got 447 / 1000 correct (44.70)
```

6.0.6 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.

[23]:
```
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(in_channel=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 = 2.6948
Checking accuracy on validation set
Got 117 / 1000 correct (11.70)

Iteration 100, loss = 1.9422
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Iteration 200, loss = 1.5531
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 300, loss = 1.4329
Checking accuracy on validation set
Got 397 / 1000 correct (39.70)

Iteration 400, loss = 1.4580
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)

Iteration 500, loss = 1.7935
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 600, loss = 1.3080
Checking accuracy on validation set
Got 451 / 1000 correct (45.10)

Iteration 700, loss = 1.4459
Checking accuracy on validation set
Got 480 / 1000 correct (48.00)
```

7 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.

7.0.1 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.

[24]:
```
# 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 = 2.3353
Checking accuracy on validation set
Got 158 / 1000 correct (15.80)

Iteration 100, loss = 1.7784
Checking accuracy on validation set
Got 377 / 1000 correct (37.70)

Iteration 200, loss = 2.1283
Checking accuracy on validation set
Got 412 / 1000 correct (41.20)

Iteration 300, loss = 1.7129
```

```
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Iteration 400, loss = 1.8840
Checking accuracy on validation set
Got 406 / 1000 correct (40.60)

Iteration 500, loss = 1.6429
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)

Iteration 600, loss = 1.5488
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)

Iteration 700, loss = 1.4457
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)
```

7.0.2 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 can use the default PyTorch weight initialization.

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.

[57]:
```
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(in_channels=3, out_channels=channel_1, kernel_size=5, padding=2,␣
  ↪bias=True),
    nn.ReLU(),
    nn.Conv2d(in_channels=channel_1, out_channels=channel_2, kernel_size=3,␣
  ↪padding=1, bias=True),
    nn.ReLU(),
    Flatten(),
    nn.Linear(in_features=32*32*channel_2, out_features=10, bias=True),
)

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 = 2.2994
Checking accuracy on validation set
Got 124 / 1000 correct (12.40)

Iteration 100, loss = 1.6464
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Iteration 200, loss = 1.2827
Checking accuracy on validation set
Got 517 / 1000 correct (51.70)

Iteration 300, loss = 1.3574
Checking accuracy on validation set
Got 516 / 1000 correct (51.60)

Iteration 400, loss = 1.1129
Checking accuracy on validation set
Got 542 / 1000 correct (54.20)

Iteration 500, loss = 1.4629
Checking accuracy on validation set
Got 549 / 1000 correct (54.90)

Iteration 600, loss = 1.2995
```

```
Checking accuracy on validation set
Got 560 / 1000 correct (56.00)

Iteration 700, loss = 1.1438
Checking accuracy on validation set
Got 601 / 1000 correct (60.10)
```

8 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

8.0.1 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.

8.0.2 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.

8.0.3 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

8.0.4 Have fun and happy training!

[53]:
```
################################################################################
# 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)*****

channel_1 = 32
channel_2 = 64
channel_3 = 128
channel_4 = 256
```

```
kerneal_size_1 = 3
kerneal_size_2 = 3
kerneal_size_3 = 3
kerneal_size_4 = 3
pool_kernel_size = 3
padding = 1

learning_rate = 1e-3

model = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=channel_1,␣
 ↪kernel_size=kerneal_size_1, padding=padding, bias=True),
    nn.BatchNorm2d(num_features=channel_1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=pool_kernel_size, padding=padding),

    nn.Conv2d(in_channels=channel_1, out_channels=channel_2,␣
 ↪kernel_size=kerneal_size_2, padding=padding, bias=True),
    nn.BatchNorm2d(num_features=channel_2),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=pool_kernel_size, padding=padding),

    nn.Conv2d(in_channels=channel_2, out_channels=channel_3,␣
 ↪kernel_size=kerneal_size_3, padding=padding, bias=True),
    nn.BatchNorm2d(num_features=channel_3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=pool_kernel_size, padding=padding),

    nn.Conv2d(in_channels=channel_3, out_channels=channel_4,␣
 ↪kernel_size=kerneal_size_4, padding=padding, bias=True),
    nn.BatchNorm2d(num_features=channel_4),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=pool_kernel_size, padding=padding),

    nn.Flatten(),
    nn.Linear(in_features=256 ,out_features=10, bias=True),
    nn.Softmax(),
)

#optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9,␣
 ↪nesterov=True)
optimizer = optim.Adam(params=model.parameters(), lr=learning_rate)
#optimizer = optim.RMSprop(params=model.parameters(), lr=1e-4, momentum=0.9)

print_every = 1e6

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

```
################################################################################
#                                END OF YOUR CODE                              #
################################################################################

# You should get at least 70% accuracy.
# You may modify the number of epochs to any number below 15.
train_part34(model, optimizer, epochs=10)
```

```
Iteration 0, loss = 2.2950
Checking accuracy on validation set
Got 153 / 1000 correct (15.30)

Iteration 0, loss = 1.7629
Checking accuracy on validation set
Got 626 / 1000 correct (62.60)

Iteration 0, loss = 1.8065
Checking accuracy on validation set
Got 669 / 1000 correct (66.90)

Iteration 0, loss = 1.7653
Checking accuracy on validation set
Got 696 / 1000 correct (69.60)

Iteration 0, loss = 1.7795
Checking accuracy on validation set
Got 696 / 1000 correct (69.60)

Iteration 0, loss = 1.6267
Checking accuracy on validation set
Got 694 / 1000 correct (69.40)

Iteration 0, loss = 1.6736
Checking accuracy on validation set
Got 714 / 1000 correct (71.40)

Iteration 0, loss = 1.7288
Checking accuracy on validation set
Got 726 / 1000 correct (72.60)

Iteration 0, loss = 1.7051
Checking accuracy on validation set
Got 758 / 1000 correct (75.80)

Iteration 0, loss = 1.6325
Checking accuracy on validation set
Got 749 / 1000 correct (74.90)
```

8.1 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.

Answer:

First, I tried 2 layers of the {conv - relu- maxpool} suggested + affine + softmax. I tried SGD, Adam, and RMSprop optimizers, and ajusted the values of the respective learning rates and/or momentum for each. I chose to continue with Adam as it was the one giving the best results during my tests.

As the validation accuracy was still too low, I added one then two layers of {conv - relu- maxpool} to have 4 layers total. I kept kernel sizes of 3 and paddings of 1 to conserve dimensions, and I ajusted the channels. I figured that having values of 2^x with increasing x at each layer gives better results.

Then, I added batch normailzation to get a validation accuracy that is greater than 70%.

8.2 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.

[54]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 7391 / 10000 correct (73.91)
```