# FullyConnectedNets

November 2, 2024

[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 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.

[3]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
```

```python
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))))
```

[4]:
```python
# 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.

[5]:
```python
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:  2.300479089758513
W1 relative error: 2.422780825313861e-07
W2 relative error: 0.00022588956117918746
W3 relative error: 1.0917357359801169e-07
b1 relative error: 7.588003489874129e-09
b2 relative error: 8.435781647563787e-10
b3 relative error: 8.538354774444772e-11
Running check with reg =  3.14
Initial loss:  7.052114776523025
W1 relative error: 4.845344814614926e-09
W2 relative error: 7.041536542395698e-08
W3 relative error: 2.2242182923626534e-08
b1 relative error: 1.4752427965311745e-08
b2 relative error: 1.786160077390948e-09
b3 relative error: 2.0285692041460677e-10
```

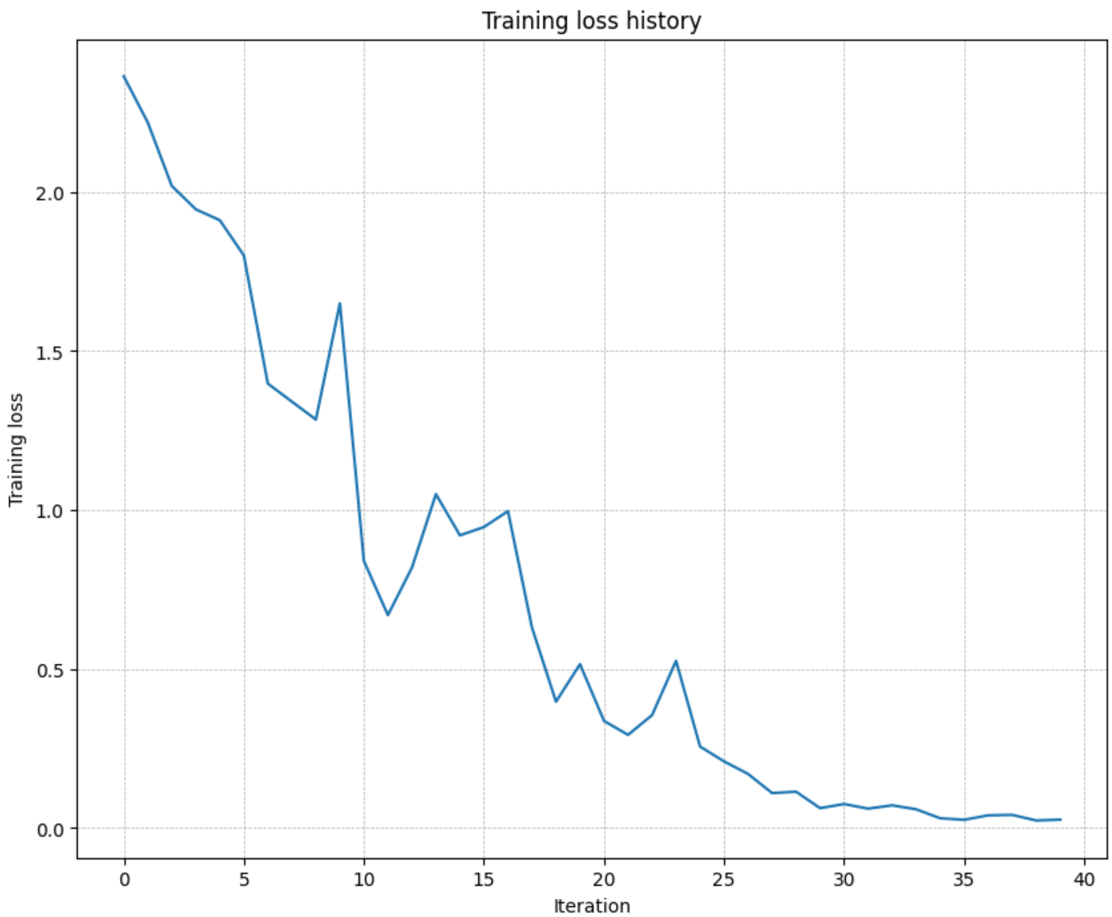
As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy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

[6]:
```
# 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"],
```

```python
}

# weight_scale = 1e-2   # Experiment with this!
# learning_rate = 1e-4  # Experiment with this!
weight_scale = 1e-2
learning_rate = 1e-2
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: 2.363364
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.108000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.127000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.172000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.184000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.181000
(Epoch 5 / 20) train acc: 0.740000; val_acc: 0.190000
(Iteration 11 / 40) loss: 0.839976
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.187000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.183000
(Epoch 8 / 20) train acc: 0.820000; val_acc: 0.177000
(Epoch 9 / 20) train acc: 0.860000; val_acc: 0.200000
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.191000
(Iteration 21 / 40) loss: 0.337174
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.189000
(Epoch 12 / 20) train acc: 0.940000; val_acc: 0.180000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.195000
```

```
(Iteration 31 / 40) loss: 0.075911
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.182000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.201000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.207000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.185000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.192000
```



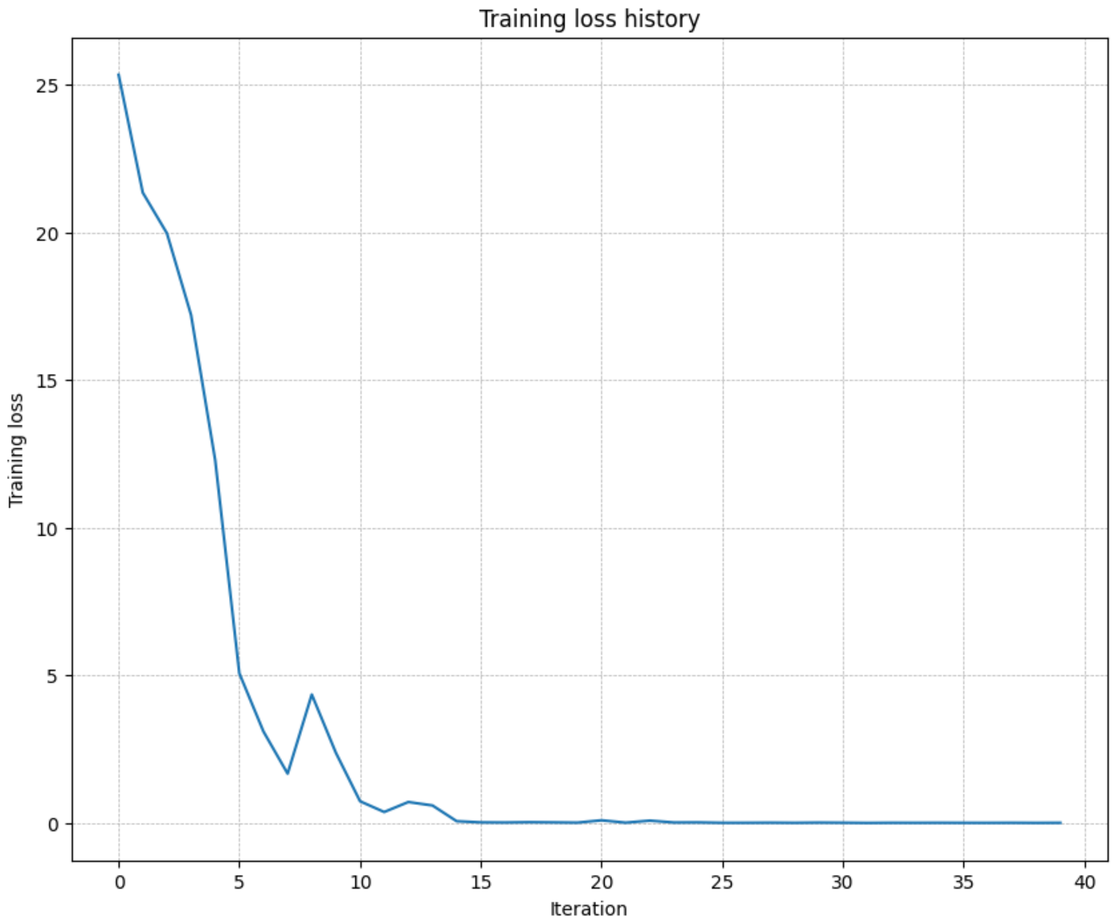

Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[7]:
```
# 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'],
}

# learning_rate = 2e-3  # Experiment with this!
# weight_scale = 1e-5   # Experiment with this!
learning_rate = 3e-3
weight_scale = 8e-2
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: 25.329444
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.111000
(Epoch 1 / 20) train acc: 0.260000; val_acc: 0.087000
(Epoch 2 / 20) train acc: 0.280000; val_acc: 0.115000
(Epoch 3 / 20) train acc: 0.540000; val_acc: 0.130000
(Epoch 4 / 20) train acc: 0.700000; val_acc: 0.133000
(Epoch 5 / 20) train acc: 0.800000; val_acc: 0.117000
(Iteration 11 / 40) loss: 0.737084
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.129000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.138000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.135000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.134000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.131000
(Iteration 21 / 40) loss: 0.092505
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.132000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.130000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.129000
```

```
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.129000
(Iteration 31 / 40) loss: 0.010478
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.130000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.127000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.131000
```




## 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:

The five-layer network seemed more sensitive to the initialization scale compared to the three-layer network. This is because deeper networks, like the five-layer network, have more layers of weights and biases that need to be optimized. If the initialization scale is too large, the activations can explode, leading to very large gradients and unstable training. Conversely, if the initialization scale is too small, the activations can vanish, leading to very small gradients and slow or stalled training. The three-layer network, being shallower, is less prone to these issues because there are fewer layers for the gradients to propagate through, making it less sensitive to the initialization scale.

# 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.

[8]:
```
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
```

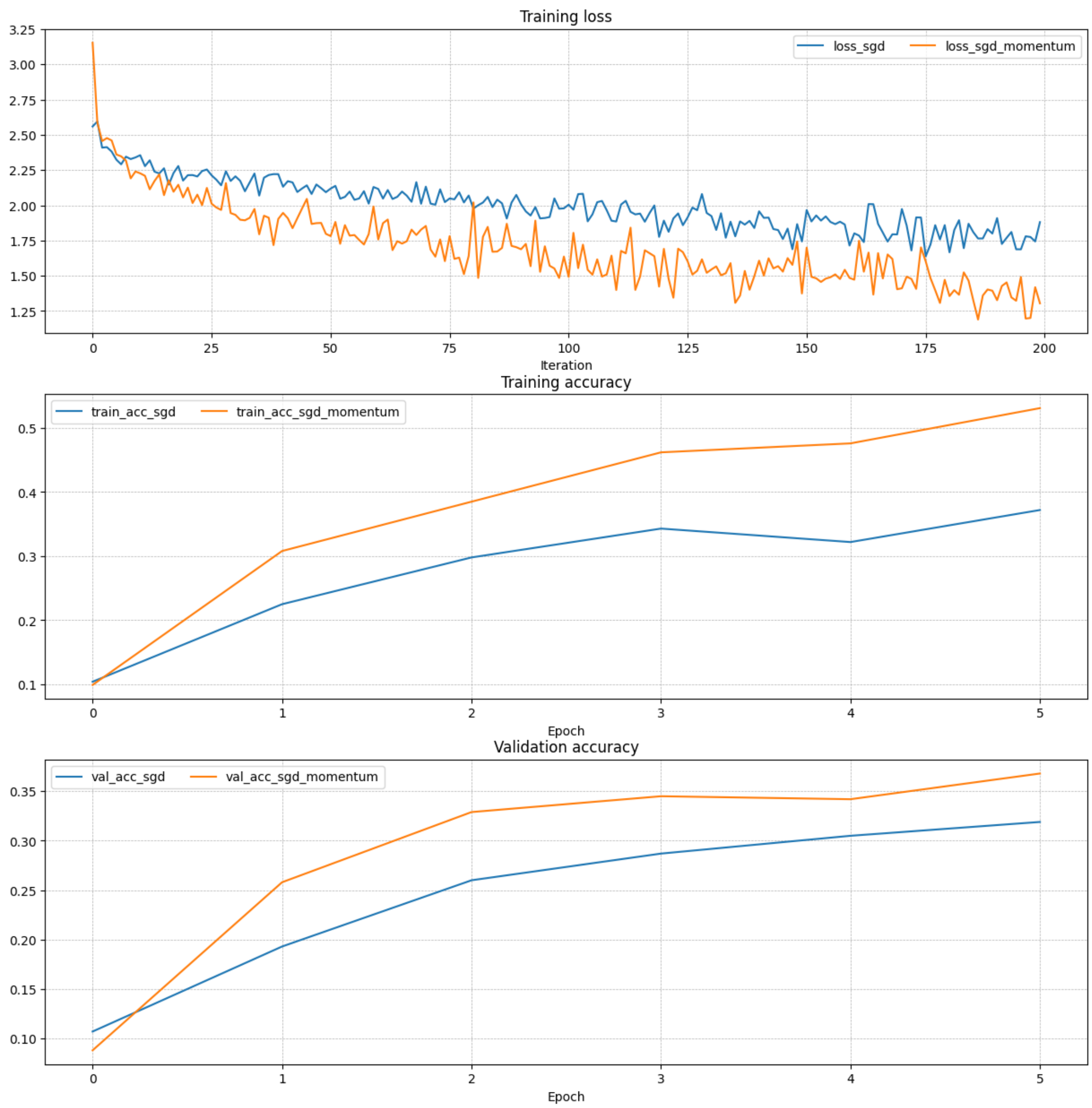
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[9]:
```
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: 2.559978
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.107000
(Iteration 11 / 200) loss: 2.356069
(Iteration 21 / 200) loss: 2.214091
(Iteration 31 / 200) loss: 2.205928
(Epoch 1 / 5) train acc: 0.225000; val_acc: 0.193000
(Iteration 41 / 200) loss: 2.132095
(Iteration 51 / 200) loss: 2.118950
(Iteration 61 / 200) loss: 2.116443
(Iteration 71 / 200) loss: 2.132549
(Epoch 2 / 5) train acc: 0.298000; val_acc: 0.260000
(Iteration 81 / 200) loss: 1.977227
(Iteration 91 / 200) loss: 2.007528
(Iteration 101 / 200) loss: 2.004762
(Iteration 111 / 200) loss: 1.885342
(Epoch 3 / 5) train acc: 0.343000; val_acc: 0.287000
(Iteration 121 / 200) loss: 1.891517
(Iteration 131 / 200) loss: 1.923677
(Iteration 141 / 200) loss: 1.957743
(Iteration 151 / 200) loss: 1.966736
(Epoch 4 / 5) train acc: 0.322000; val_acc: 0.305000
(Iteration 161 / 200) loss: 1.801483
(Iteration 171 / 200) loss: 1.973779
(Iteration 181 / 200) loss: 1.666573
(Iteration 191 / 200) loss: 1.909494
(Epoch 5 / 5) train acc: 0.372000; val_acc: 0.319000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153777
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.088000
(Iteration 11 / 200) loss: 2.227203
(Iteration 21 / 200) loss: 2.125706
(Iteration 31 / 200) loss: 1.932679
(Epoch 1 / 5) train acc: 0.308000; val_acc: 0.258000
(Iteration 41 / 200) loss: 1.946330
(Iteration 51 / 200) loss: 1.781856
(Iteration 61 / 200) loss: 1.757563
(Iteration 71 / 200) loss: 1.853951
(Epoch 2 / 5) train acc: 0.385000; val_acc: 0.329000
(Iteration 81 / 200) loss: 2.020635
(Iteration 91 / 200) loss: 1.688374
(Iteration 101 / 200) loss: 1.492405
(Iteration 111 / 200) loss: 1.399368
```

```
(Epoch 3 / 5) train acc: 0.462000; val_acc: 0.345000
(Iteration 121 / 200) loss: 1.691196
(Iteration 131 / 200) loss: 1.545281
(Iteration 141 / 200) loss: 1.607481
(Iteration 151 / 200) loss: 1.699787
(Epoch 4 / 5) train acc: 0.476000; val_acc: 0.342000
(Iteration 161 / 200) loss: 1.472431
(Iteration 171 / 200) loss: 1.411654
(Iteration 181 / 200) loss: 1.355995
(Iteration 191 / 200) loss: 1.327029
(Epoch 5 / 5) train acc: 0.531000; val_acc: 0.368000
```



### 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.

[10]:
```
# 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
```

[11]:
```
# 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
```

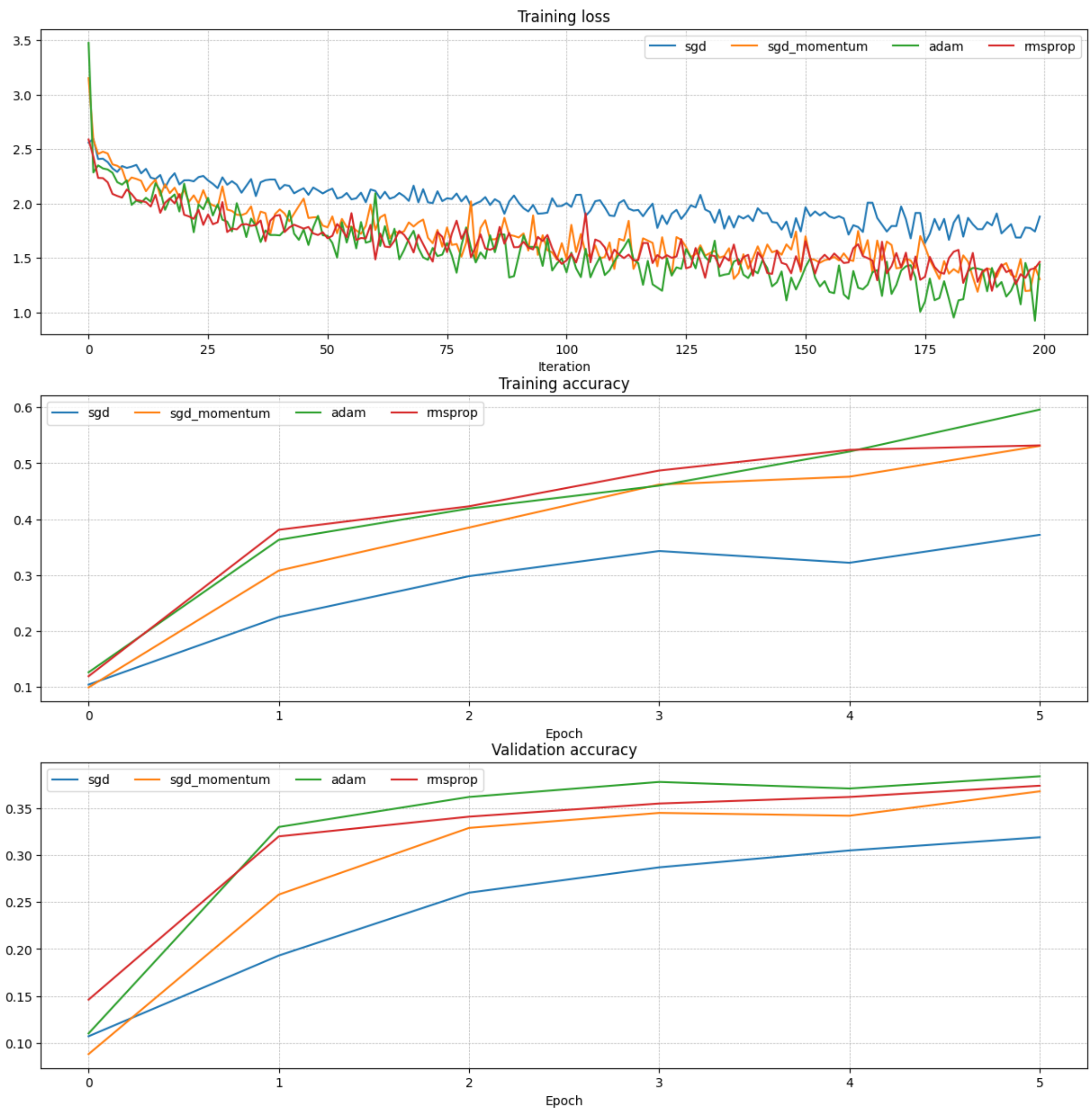
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[12]:
```
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: 3.476928
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.027712

(Iteration 21 / 200) loss: 2.183358
(Iteration 31 / 200) loss: 1.744257
(Epoch 1 / 5) train acc: 0.363000; val_acc: 0.330000
(Iteration 41 / 200) loss: 1.707951
(Iteration 51 / 200) loss: 1.703835
(Iteration 61 / 200) loss: 2.094758
(Iteration 71 / 200) loss: 1.505558
(Epoch 2 / 5) train acc: 0.419000; val_acc: 0.362000
(Iteration 81 / 200) loss: 1.594430
(Iteration 91 / 200) loss: 1.519017
(Iteration 101 / 200) loss: 1.368522
(Iteration 111 / 200) loss: 1.470401
(Epoch 3 / 5) train acc: 0.460000; val_acc: 0.378000
(Iteration 121 / 200) loss: 1.199064
(Iteration 131 / 200) loss: 1.464705
```

```
(Iteration 141 / 200) loss: 1.359863
(Iteration 151 / 200) loss: 1.417470
(Epoch 4 / 5) train acc: 0.521000; val_acc: 0.371000
(Iteration 161 / 200) loss: 1.380564
(Iteration 171 / 200) loss: 1.384368
(Iteration 181 / 200) loss: 1.123532
(Iteration 191 / 200) loss: 1.232268
(Epoch 5 / 5) train acc: 0.596000; val_acc: 0.384000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895732
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.486923
(Iteration 71 / 200) loss: 1.628511
(Epoch 2 / 5) train acc: 0.423000; val_acc: 0.341000
(Iteration 81 / 200) loss: 1.506182
(Iteration 91 / 200) loss: 1.600674
(Iteration 101 / 200) loss: 1.478501
(Iteration 111 / 200) loss: 1.577708
(Epoch 3 / 5) train acc: 0.487000; val_acc: 0.355000
(Iteration 121 / 200) loss: 1.495931
(Iteration 131 / 200) loss: 1.525799
(Iteration 141 / 200) loss: 1.552580
(Iteration 151 / 200) loss: 1.654283
(Epoch 4 / 5) train acc: 0.524000; val_acc: 0.362000
(Iteration 161 / 200) loss: 1.589372
(Iteration 171 / 200) loss: 1.413528
(Iteration 181 / 200) loss: 1.500273
(Iteration 191 / 200) loss: 1.365942
(Epoch 5 / 5) train acc: 0.532000; val_acc: 0.374000
```




## 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:

The updates in AdaGrad become very small because the cache term accumulates the squared gradients over time. As training progresses, the cache term grows larger, causing the denominator np.sqrt(cache) + eps to increase. This results in smaller updates to the weights w, which can slow down learning significantly, especially in later stages of training.

Adam, on the other hand, addresses this issue by using a moving average of the squared gradients (and also the gradients themselves). This moving average prevents the cache term from growing indefinitely, allowing for more consistent updates throughout training. Therefore, Adam does not suffer from the same issue of diminishing updates as AdaGrad does.

## 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.

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

# JIANTENG
best_val = -1
best_model = None

large_data = {
    'X_train': data['X_train'],
    'y_train': data['y_train'],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

learning_rates = np.linspace(1e-3, 1e-2, num=3)
regularization_strengths = np.linspace(1e-3, 1e-2, num=3)
weight_scales = np.linspace(1e-2, 1e-1, num=5)
```

```
params = [(lr, reg, ws) for lr in learning_rates for reg in␣
 ↪regularization_strengths for ws in weight_scales]

for (lr, reg, ws) in params:
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=ws,
        reg=reg
    )
    solver = Solver(
        model,
        data,
        num_epochs=20,
        batch_size=200,
        update_rule='adam',
        optim_config={'learning_rate': lr},
        verbose=False
    )
    solver.train()
    val_acc = solver.best_val_acc
    print(f"Validation accuracy: {val_acc} with lr={lr}, reg={reg}, ws={ws}")
    if val_acc > best_val:
        best_val = val_acc
        best_model = model

print(f"Best validation accuracy: {best_val}")

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

```
Validation accuracy: 0.526 with lr=0.001, reg=0.001, ws=0.01
Validation accuracy: 0.538 with lr=0.001, reg=0.001, ws=0.0325
Validation accuracy: 0.54 with lr=0.001, reg=0.001, ws=0.05500000000000001
Validation accuracy: 0.527 with lr=0.001, reg=0.001, ws=0.0775
Validation accuracy: 0.512 with lr=0.001, reg=0.001, ws=0.1
Validation accuracy: 0.119 with lr=0.001, reg=0.0055000000000000005, ws=0.01
Validation accuracy: 0.537 with lr=0.001, reg=0.0055000000000000005, ws=0.0325
Validation accuracy: 0.531 with lr=0.001, reg=0.0055000000000000005,
ws=0.05500000000000001
Validation accuracy: 0.53 with lr=0.001, reg=0.0055000000000000005, ws=0.0775
Validation accuracy: 0.514 with lr=0.001, reg=0.0055000000000000005, ws=0.1
Validation accuracy: 0.119 with lr=0.001, reg=0.01, ws=0.01
Validation accuracy: 0.519 with lr=0.001, reg=0.01, ws=0.0325
Validation accuracy: 0.531 with lr=0.001, reg=0.01, ws=0.05500000000000001
Validation accuracy: 0.511 with lr=0.001, reg=0.01, ws=0.0775
```

```
Validation accuracy: 0.51 with lr=0.001, reg=0.01, ws=0.1
Validation accuracy: 0.437 with lr=0.0055000000000000005, reg=0.001, ws=0.01
Validation accuracy: 0.44 with lr=0.0055000000000000005, reg=0.001, ws=0.0325
Validation accuracy: 0.44 with lr=0.0055000000000000005, reg=0.001,
ws=0.05500000000000001
Validation accuracy: 0.47 with lr=0.0055000000000000005, reg=0.001, ws=0.0775
Validation accuracy: 0.491 with lr=0.0055000000000000005, reg=0.001, ws=0.1
Validation accuracy: 0.119 with lr=0.0055000000000000005,
reg=0.0055000000000000005, ws=0.01
Validation accuracy: 0.399 with lr=0.0055000000000000005,
reg=0.0055000000000000005, ws=0.0325
Validation accuracy: 0.409 with lr=0.0055000000000000005,
reg=0.0055000000000000005, ws=0.05500000000000001
Validation accuracy: 0.426 with lr=0.0055000000000000005,
reg=0.0055000000000000005, ws=0.0775
Validation accuracy: 0.461 with lr=0.0055000000000000005,
reg=0.0055000000000000005, ws=0.1
Validation accuracy: 0.119 with lr=0.0055000000000000005, reg=0.01, ws=0.01
Validation accuracy: 0.395 with lr=0.0055000000000000005, reg=0.01, ws=0.0325
Validation accuracy: 0.391 with lr=0.0055000000000000005, reg=0.01,
ws=0.05500000000000001
Validation accuracy: 0.432 with lr=0.0055000000000000005, reg=0.01, ws=0.0775
Validation accuracy: 0.449 with lr=0.0055000000000000005, reg=0.01, ws=0.1
Validation accuracy: 0.312 with lr=0.01, reg=0.001, ws=0.01
Validation accuracy: 0.345 with lr=0.01, reg=0.001, ws=0.0325
Validation accuracy: 0.351 with lr=0.01, reg=0.001, ws=0.05500000000000001
Validation accuracy: 0.422 with lr=0.01, reg=0.001, ws=0.0775
Validation accuracy: 0.473 with lr=0.01, reg=0.001, ws=0.1
Validation accuracy: 0.119 with lr=0.01, reg=0.0055000000000000005, ws=0.01
Validation accuracy: 0.325 with lr=0.01, reg=0.0055000000000000005, ws=0.0325
Validation accuracy: 0.364 with lr=0.01, reg=0.0055000000000000005,
ws=0.05500000000000001
Validation accuracy: 0.386 with lr=0.01, reg=0.0055000000000000005, ws=0.0775
Validation accuracy: 0.428 with lr=0.01, reg=0.0055000000000000005, ws=0.1
Validation accuracy: 0.119 with lr=0.01, reg=0.01, ws=0.01
Validation accuracy: 0.319 with lr=0.01, reg=0.01, ws=0.0325
Validation accuracy: 0.344 with lr=0.01, reg=0.01, ws=0.05500000000000001
Validation accuracy: 0.36 with lr=0.01, reg=0.01, ws=0.0775
Validation accuracy: 0.419 with lr=0.01, reg=0.01, ws=0.1
Best validation accuracy: 0.54
```

## 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.

[14]:

```
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:  0.54
Test set accuracy:  0.512
```

# BatchNormalization

November 2, 2024

[ ]:
```
# 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.

[73]:
```
# 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
```

[2]:
```
# 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!

[3]:
```python
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

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

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

gamma = np.ones((D3,))
beta = np.zeros((D3,))

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

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])

# Now means should be close to beta and stds close to gamma.
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)
```

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

```
After batch normalization (gamma=1, beta=0)
  means: [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]
```

[4]:

```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.

[51]:
```python
# 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:  1.7029244130916756e-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 $\mu$ and variance $v$. With $\mu$ and $v$ calculated, we can calculate the standard deviation $\sigma$ 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.

[46]:
```
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:  0.0
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 3.13x
```

## 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`.

[54]:
```
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:  2.261195510124443
W1 relative error: 1.10e-04
W2 relative error: 5.65e-06
```

```
W3 relative error: 3.75e-10
b1 relative error: 2.66e-07
b2 relative error: 4.44e-08
b3 relative error: 1.01e-10
beta1 relative error: 7.85e-09
beta2 relative error: 1.07e-09
gamma1 relative error: 6.96e-09
gamma2 relative error: 2.41e-09

Running check with reg =  3.14
Initial loss:  6.996533220098718
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 5.60e-09
b1 relative error: 1.38e-08
b2 relative error: 7.99e-07
b3 relative error: 2.23e-10
beta1 relative error: 6.65e-09
beta2 relative error: 5.69e-09
gamma1 relative error: 6.60e-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.

[55]:
```
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:')
```

```python
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()
```

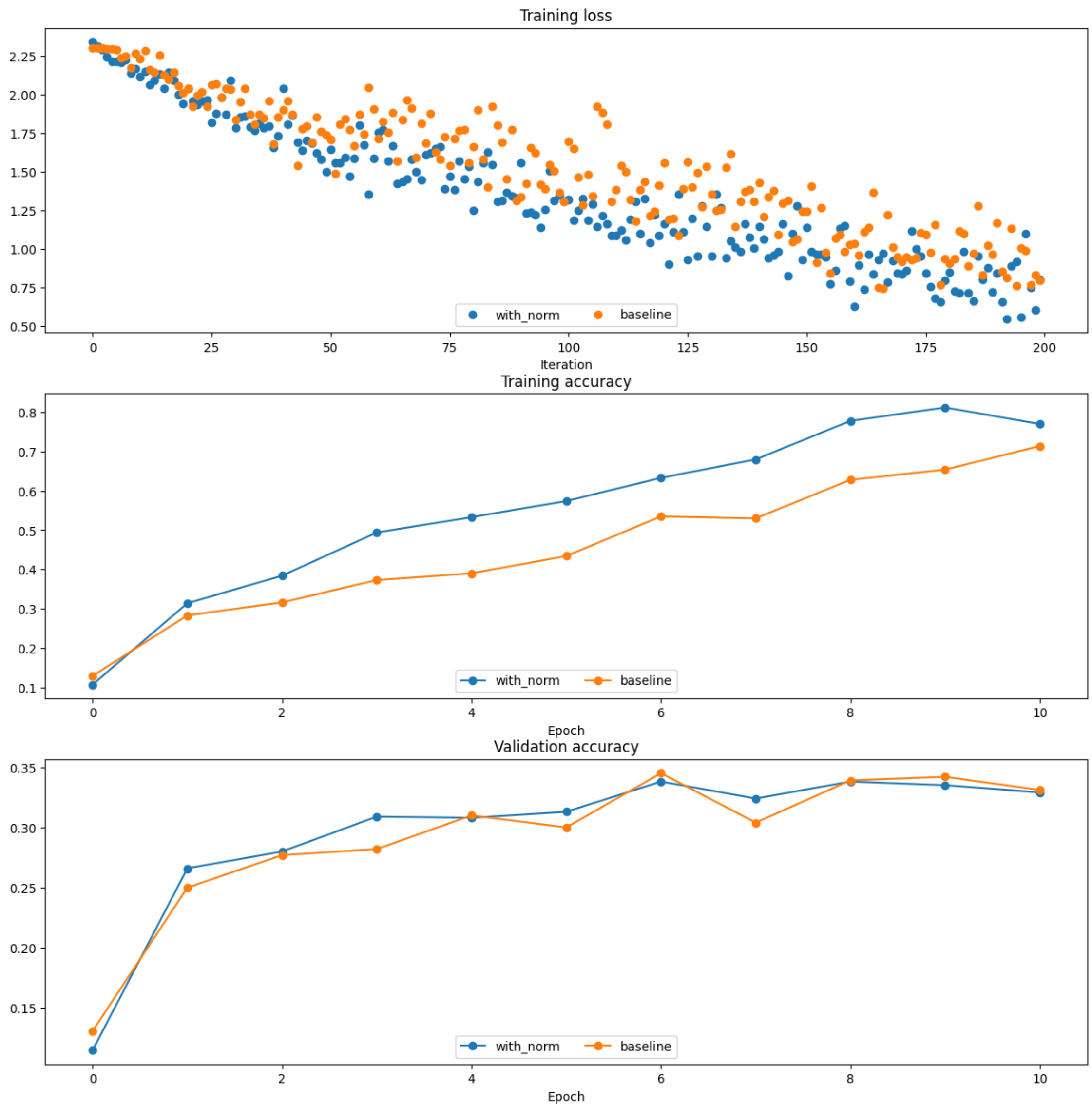
```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340975
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.314000; val_acc: 0.266000
(Iteration 21 / 200) loss: 2.039365
(Epoch 2 / 10) train acc: 0.384000; val_acc: 0.280000
(Iteration 41 / 200) loss: 2.041103
(Epoch 3 / 10) train acc: 0.494000; val_acc: 0.309000
(Iteration 61 / 200) loss: 1.753903
(Epoch 4 / 10) train acc: 0.533000; val_acc: 0.308000
(Iteration 81 / 200) loss: 1.246584
(Epoch 5 / 10) train acc: 0.574000; val_acc: 0.313000
(Iteration 101 / 200) loss: 1.320589
(Epoch 6 / 10) train acc: 0.633000; val_acc: 0.338000
(Iteration 121 / 200) loss: 1.159472
(Epoch 7 / 10) train acc: 0.680000; val_acc: 0.324000
(Iteration 141 / 200) loss: 1.146092
(Epoch 8 / 10) train acc: 0.778000; val_acc: 0.338000
(Iteration 161 / 200) loss: 0.627672
(Epoch 9 / 10) train acc: 0.812000; val_acc: 0.335000
(Iteration 181 / 200) loss: 0.848866
(Epoch 10 / 10) train acc: 0.770000; val_acc: 0.329000

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: 1.696058
(Epoch 6 / 10) train acc: 0.535000; val_acc: 0.345000
(Iteration 121 / 200) loss: 1.557986
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.304000
(Iteration 141 / 200) loss: 1.432189
(Epoch 8 / 10) train acc: 0.628000; val_acc: 0.339000
(Iteration 161 / 200) loss: 1.034117
(Epoch 9 / 10) train acc: 0.654000; val_acc: 0.342000
(Iteration 181 / 200) loss: 0.905796
(Epoch 10 / 10) train acc: 0.714000; val_acc: 0.331000
```

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.

[56]:
```
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.

[57]:
```
np.random.seed(231)

# Try training a very deep net with batchnorm.
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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

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

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

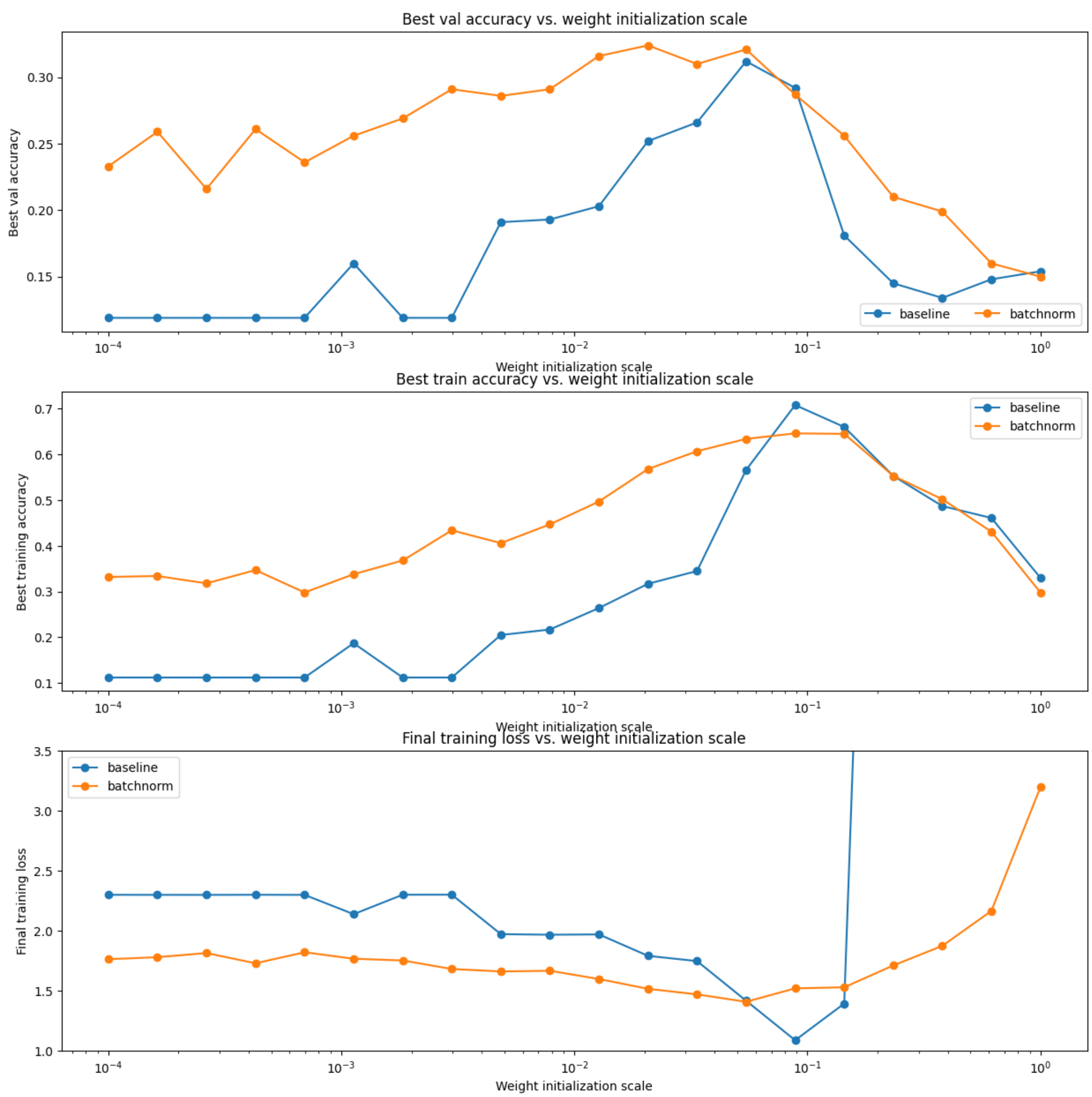
```
Running weight scale 1 / 20
Running weight scale 2 / 20
```

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

[58]:

```
# 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:

1. Best Validation Accuracy:
    - Models with batch normalization (orange line) consistently achieve higher validation accuracy across a wide range of weight initialization scales, while models without batch normalization (baseline, blue line) show lower and more variable validation accuracy.
    - For models without batch normalization, the validation accuracy improves gradually as the weight initialization scale increases but peaks at around $10^{-1}$, after which it declines sharply.
    - Models with batch normalization maintain relatively high validation accuracy even at higher initialization scales, indicating greater robustness to initialization scale.
2. Best Training Accuracy:
    - Similar to the validation accuracy, models with batch normalization reach higher training accuracy across various initialization scales.
    - For models without batch normalization, training accuracy increases as the initialization scale grows but also peaks around $10^{-1}$ and then decreases.
    - Batch normalization helps the model achieve more stable training accuracy across different weight initializations, while the baseline changes sharper with more instability.
3. Final Training Loss:
    - Models with batch normalization show lower final training loss across most initialization scales, while models without batch normalization have higher and more fluctuating final training loss values.
    - The final training loss for models without batch normalization sharply increases when the initialization scale is around $10^{-1}$, highlighting instability without normalization.

Batch normalization helps stabilize the training process, allowing models to tolerate a wider range of weight initialization scales. Without batch normalization, models are sensitive to initialization scale: too small scales result in slow convergence, while too large scales cause instability and poor performance. Batch normalization mitigates these issues by normalizing the activations within each batch, reducing internal covariate shift, and thus making training less dependent on careful weight initialization.

# 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.

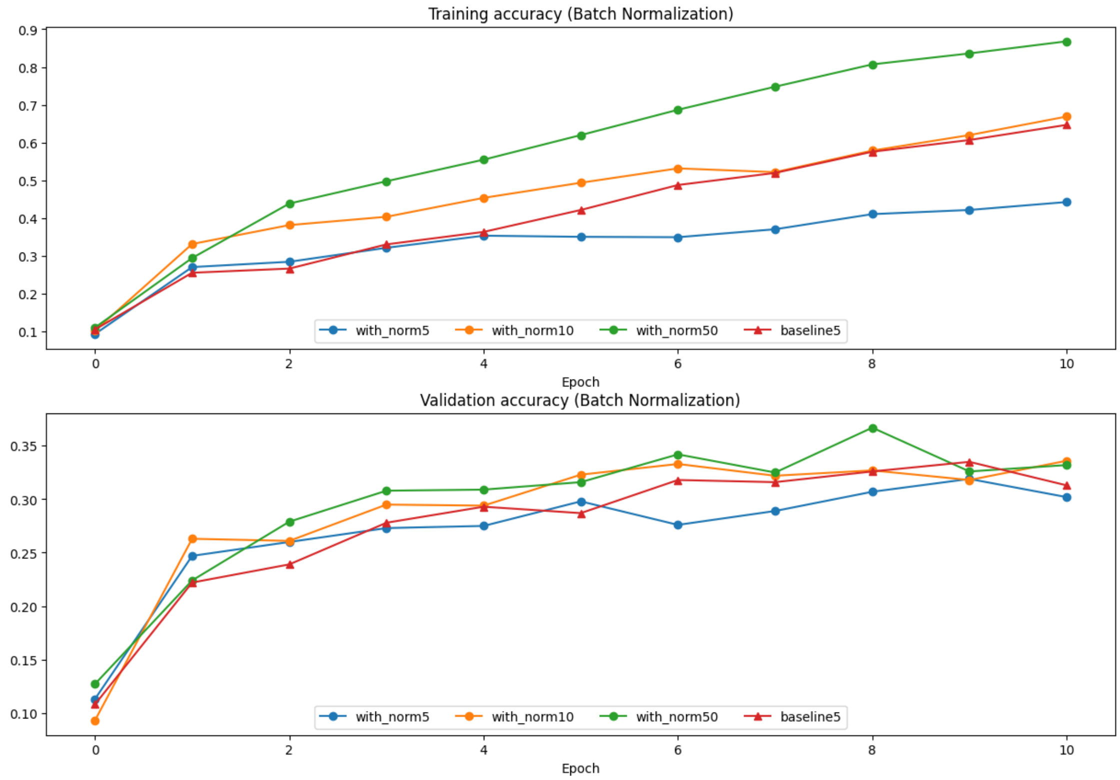
[59]:
```
def run_batchsize_experiments(normalization_mode):
    np.random.seed(231)
```

```
    # 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
```

[78]:
```
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:

This experiment investigates the effect of batch size on training and validation accuracy for models with and without batch normalization. The batch sizes are represented by different labels in the legend: `with_norm5`, `with_norm10`, `with_norm50` (indicating models with batch normalization and varying batch sizes), and `baseline5` (a baseline model with batch size 5 but without batch normalization).

1. Training Accuracy:
    - The model with batch normalization and a larger batch size (`with_norm50`) shows the highest training accuracy, consistently outperforming the models with smaller batch sizes (`with_norm10` and `with_norm5`) and the baseline model.
    - Smaller batch sizes with batch normalization (`with_norm5` and `with_norm10`) result in lower training accuracy, with `with_norm5` performing the worst among all models.
    - This suggests that larger batch sizes improve the effectiveness of batch normalization, leading to better training performance.
2. Validation Accuracy:
    - While validation accuracy generally improves with batch size, the difference between batch sizes is less pronounced than in training accuracy. The model with the largest batch size (`with_norm50`) still achieves relatively better validation accuracy, but the improvement is less stable across epochs.
    - The baseline model (`baseline5`) performs similarly to smaller batch sizes (`with_norm5` and `with_norm10`) in terms of validation accuracy, indicating that smaller batch sizes may limit the generalization benefits of batch normalization.

The relationship between batch normalization and batch size shows that larger batch sizes are generally more beneficial for models with batch normalization. This is because batch normalization calculates the mean and variance of activations over each batch; larger batch sizes provide more reliable statistics, leading to more stable normalization and improved training. With smaller batch sizes, the statistics can vary significantly across batches, making normalization less effective and potentially leading to poorer convergence and generalization.

## 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:

Batch normalization: None of the options.

Layer normalization: Option `2`.

1. Option `1` involves scaling each image so that the RGB channels for each row of pixels within an image sum up to 1. This is not directly analogous to either batch normalization or layer normalization. It normalizes the data on a per-row basis within each image, which is different from the per-feature normalization in batch normalization and the per-data-point normalization in layer normalization.

2. Option `2` is analogous to layer normalization. Layer normalization involves normalizing the activations of a layer across all features for each individual data point, ensuring that the mean activation is 0 and the variance is 1 for each data point. Scaling each image so that the RGB channels for all pixels within an image sum up to 1 is similar to normalizing across all features within a single data point.

3. Option `3` is kind of analogous to batch normalization because batch normalization involves normalizing the activations of a layer across the entire batch of data, ensuring that the mean activation is 0 and the variance is 1 for each feature across the batch. Subtracting the mean image of the dataset from each image in the dataset only centers the data but does not normalize the variance, and it is done on a per-image basis rather than across a batch of images. Batch normalization also includes a scaling step after normalization, which is not present in option 3.

4. Option `4` involves setting all RGB values to either 0 or 1 depending on a given threshold. This is a form of binarization and is not analogous to either batch normalization or layer normalization. Binarization is a different type of preprocessing that does not involve normalizing the mean and variance of the data.

## 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.

[70]:
```python
# 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]
```

[71]:
```
# 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:  1.4336159772435053e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12
```

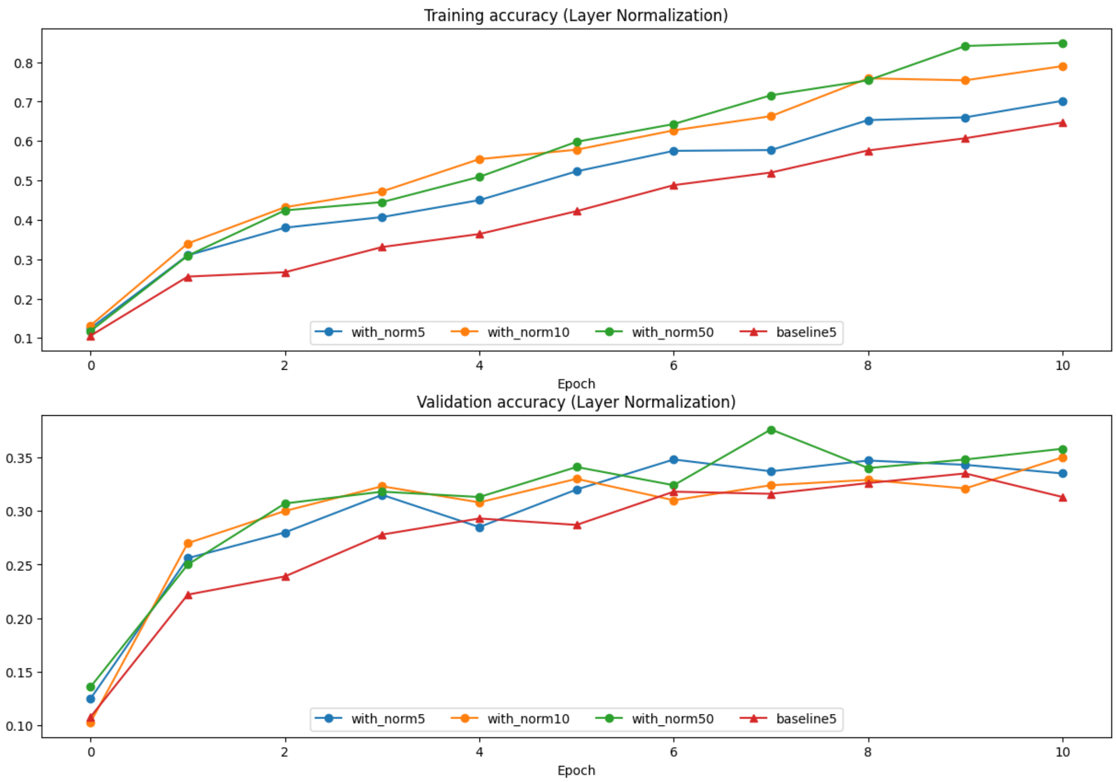
# 11 Layer Normalization and Batch Size

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!

[77]:
```
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:

The answer is `2. Having a very small dimension of features`. Layer normalization normalizes the activations across all features for each individual data point. When the dimension of features is very small, the statistics (mean and variance) computed for normalization may not be stable or representative, leading to poor performance. This instability can cause the normalization process to be less effective, potentially harming the training process and the overall performance of the network.

In contrast, in case `1. Using it in a very deep network`, layer normalization can still be effective, as it normalizes across features within each layer, helping to stabilize the training process. Case `3. Having a high regularization term` affects the overall training process by penalizing large weights, but it does not directly impact the effectiveness of layer normalization.

# Dropout

November 2, 2024

[ ]:
```
# 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

[1]:
```
# 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))))
```

```
=========== You can safely ignore the message below if you are NOT working on
ConvolutionalNetworks.ipynb ===========
        You will need to compile a Cython extension for a portion of this
assignment.
        The instructions to do this will be given in a section of the notebook
below.
```

[2]:
```
# 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.

[3]:
```
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.

[4]:
```
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:

If we do not divide the values being passed through inverse dropout by p in the dropout layer, the expected value of the activations will be scaled down by a factor of p during training. This happens because dropout randomly sets a fraction p of the activations to zero, and without scaling, the remaining activations are not adjusted to maintain the same expected value.

# 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.

[5]:
```python
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:  2.300479089758513
W1 relative error: 2.42e-07
W2 relative error: 2.26e-04
W3 relative error: 1.09e-07
b1 relative error: 7.59e-09
b2 relative error: 8.44e-10
b3 relative error: 8.54e-11

Running check with dropout =  0.75
Initial loss:  2.301648215765084
W1 relative error: 8.34e-08
W2 relative error: 1.48e-06
W3 relative error: 1.24e-07
b1 relative error: 7.66e-09
b2 relative error: 1.79e-09
b3 relative error: 9.27e-11

Running check with dropout =  0.5
Initial loss:  2.294963257966158
W1 relative error: 5.79e-08
W2 relative error: 4.66e-07
W3 relative error: 3.75e-07
b1 relative error: 4.45e-09
b2 relative error: 1.02e-08
b3 relative error: 1.19e-10
```

## 5 Regularization Experiment

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

[10]:
```
# 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 = {}
```

```python
dropout_choices = [1, 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()
```

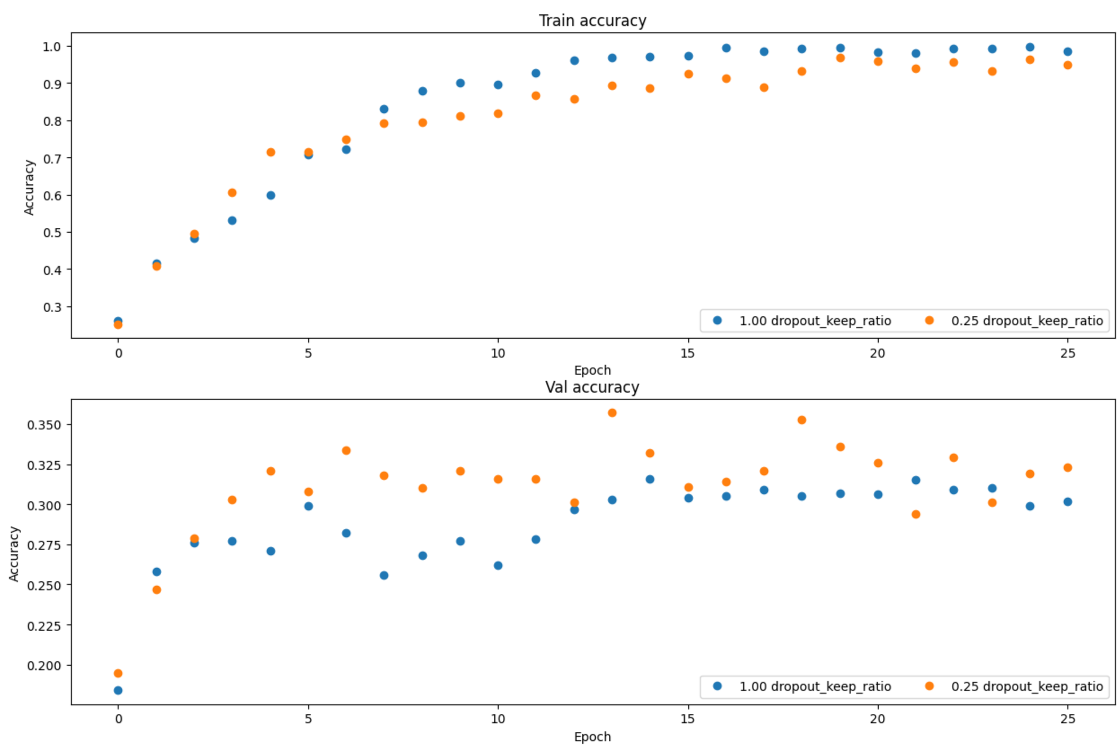
```
1
(Iteration 1 / 125) loss: 7.856204
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.256000
(Epoch 8 / 25) train acc: 0.878000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.277000
(Epoch 10 / 25) train acc: 0.896000; val_acc: 0.262000
(Epoch 11 / 25) train acc: 0.928000; val_acc: 0.278000
(Epoch 12 / 25) train acc: 0.962000; val_acc: 0.297000
(Epoch 13 / 25) train acc: 0.968000; val_acc: 0.303000
(Epoch 14 / 25) train acc: 0.972000; val_acc: 0.316000
(Epoch 15 / 25) train acc: 0.974000; val_acc: 0.304000
(Epoch 16 / 25) train acc: 0.996000; val_acc: 0.305000
(Epoch 17 / 25) train acc: 0.986000; val_acc: 0.309000
(Epoch 18 / 25) train acc: 0.992000; val_acc: 0.305000
(Epoch 19 / 25) train acc: 0.994000; val_acc: 0.307000
(Epoch 20 / 25) train acc: 0.982000; val_acc: 0.306000
(Iteration 101 / 125) loss: 0.134114
(Epoch 21 / 25) train acc: 0.980000; val_acc: 0.315000
(Epoch 22 / 25) train acc: 0.992000; val_acc: 0.309000
```

```
(Epoch 23 / 25) train acc: 0.992000; val_acc: 0.310000
(Epoch 24 / 25) train acc: 0.998000; val_acc: 0.299000
(Epoch 25 / 25) train acc: 0.986000; val_acc: 0.302000

0.25
(Iteration 1 / 125) loss: 15.575763
(Epoch 0 / 25) train acc: 0.252000; val_acc: 0.195000
(Epoch 1 / 25) train acc: 0.408000; val_acc: 0.247000
(Epoch 2 / 25) train acc: 0.494000; val_acc: 0.279000
(Epoch 3 / 25) train acc: 0.606000; val_acc: 0.303000
(Epoch 4 / 25) train acc: 0.714000; val_acc: 0.321000
(Epoch 5 / 25) train acc: 0.714000; val_acc: 0.308000
(Epoch 6 / 25) train acc: 0.748000; val_acc: 0.334000
(Epoch 7 / 25) train acc: 0.792000; val_acc: 0.318000
(Epoch 8 / 25) train acc: 0.794000; val_acc: 0.310000
(Epoch 9 / 25) train acc: 0.812000; val_acc: 0.321000
(Epoch 10 / 25) train acc: 0.818000; val_acc: 0.316000
(Epoch 11 / 25) train acc: 0.866000; val_acc: 0.316000
(Epoch 12 / 25) train acc: 0.858000; val_acc: 0.301000
(Epoch 13 / 25) train acc: 0.894000; val_acc: 0.357000
(Epoch 14 / 25) train acc: 0.886000; val_acc: 0.332000
(Epoch 15 / 25) train acc: 0.924000; val_acc: 0.311000
(Epoch 16 / 25) train acc: 0.914000; val_acc: 0.314000
(Epoch 17 / 25) train acc: 0.888000; val_acc: 0.321000
(Epoch 18 / 25) train acc: 0.932000; val_acc: 0.353000
(Epoch 19 / 25) train acc: 0.968000; val_acc: 0.336000
(Epoch 20 / 25) train acc: 0.958000; val_acc: 0.326000
(Iteration 101 / 125) loss: 3.885983
(Epoch 21 / 25) train acc: 0.940000; val_acc: 0.294000
(Epoch 22 / 25) train acc: 0.956000; val_acc: 0.329000
(Epoch 23 / 25) train acc: 0.932000; val_acc: 0.301000
(Epoch 24 / 25) train acc: 0.964000; val_acc: 0.319000
(Epoch 25 / 25) train acc: 0.950000; val_acc: 0.323000
```

[11]:

```
# 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:

The results suggest that dropout acts as an effective regularizer by reducing overfitting and improving the generalization of the model.

### 5.2.1 With Dropout (Dropout keep ratio: 0.25):

- **Training Accuracy**: The training accuracy is lower compared to the model without dropout. This is because dropout randomly deactivates neurons during training, which prevents the model from overfitting to the training data.
- **Validation Accuracy**: The validation accuracy is higher compared to the model without dropout. This indicates better generalization to unseen data, as dropout helps to prevent overfitting by introducing noise during training.

### 5.2.2 Without Dropout (Dropout keep ratio: 1):

- **Training Accuracy**: The training accuracy is higher because the model can fully utilize all neurons during training, leading to better performance on the training data.
- **Validation Accuracy**: The validation accuracy is lower compared to the model with dropout. This suggests that the model may be overfitting to the training data and not generalizing well to unseen data.

# ConvolutionalNetworks

November 2, 2024

[1]:
```
# # 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.

[2]:
```
# 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))))
```

[3]:
```
# 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

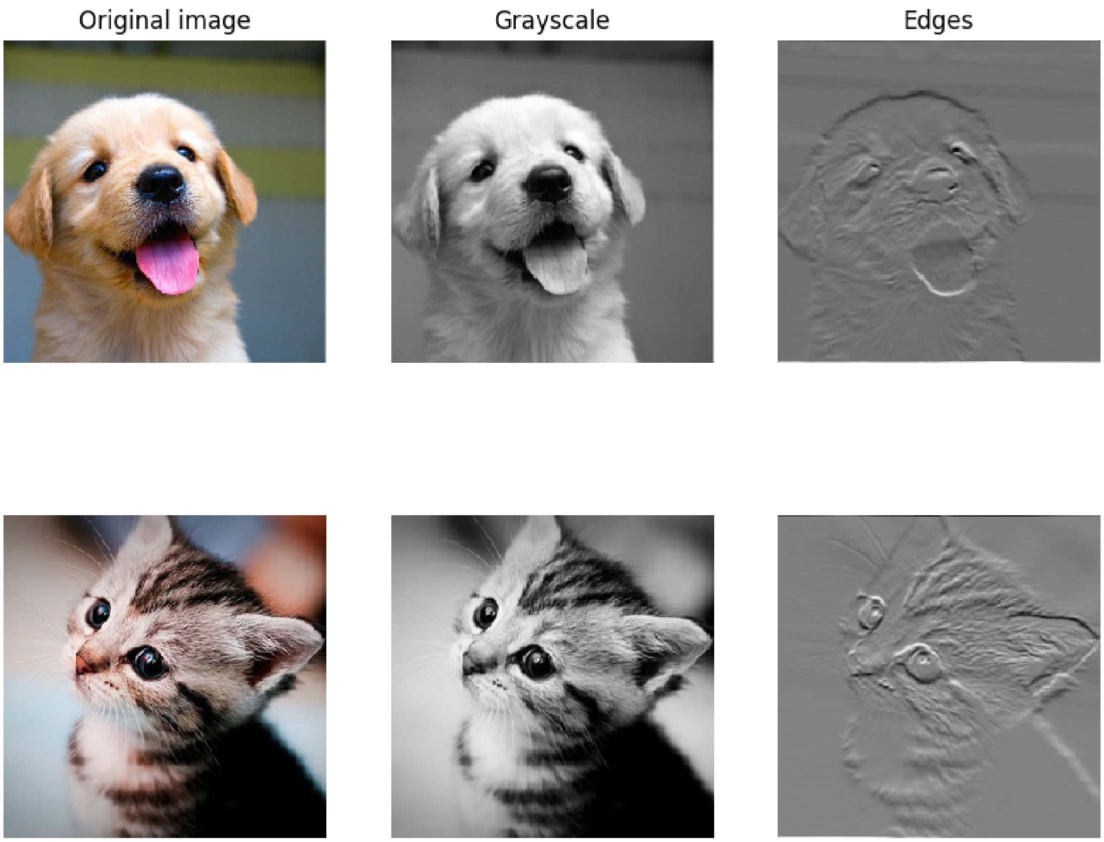
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). 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()
```

```
/tmp/ipykernel_161915/3128955772.py:4: DeprecationWarning: Starting with ImageIO
v3 the behavior of this function will switch to that of iio.v3.imread. To keep
the current behavior (and make this warning disappear) use `import imageio.v2 as
```

```
imageio` or call `imageio.v2.imread` directly.
  kitten = imread('cs231n/notebook_images/kitten.jpg')
/tmp/ipykernel_161915/3128955772.py:5: DeprecationWarning: Starting with ImageIO
v3 the behavior of this function will switch to that of iio.v3.imread. To keep
the current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  puppy = imread('cs231n/notebook_images/puppy.jpg')
```




## 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.

[14]:
```
# # 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/

%cd /data15/chenjt/ELEC4240/assignment2/cs231n/
!python setup.py build_ext --inplace
%cd ..
```

```
/data15/chenjt/ELEC4240/assignment2/cs231n
/data15/chenjt/ELEC4240/assignment2
```

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:

[9]:
```
# 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: 15.617332s
Fast: 0.168123s
Speedup: 92.892428x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 19.743929s
Fast: 0.165907s
Speedup: 119.005731x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  3.481354613192702e-14
```

[10]:
```
# 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))
```

```python
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: 0.898315s
fast: 0.004674s
speedup: 192.186075x
difference:  0.0

Testing pool_backward_fast:
Naive: 2.308027s
fast: 0.013749s
speedup: 167.873027x
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.

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

```python
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:  5.802391137330214e-09
db error:  1.0146343411762047e-09
```

[13]:
```python
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:  2.702022646099404e-10
db error:  1.451272393591721e-10
```

## 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.

[15]:
```
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):  2.30258318666065
Initial loss (with regularization):  2.508342473541143
```

### 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.

[16]:
```
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: 3.053965e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 1.005197e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.517459e-03
b3 max relative error: 1.212754e-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.

[17]:
```
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
```

[18]:
```
# Print final training accuracy.
print(
    "Small data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
```

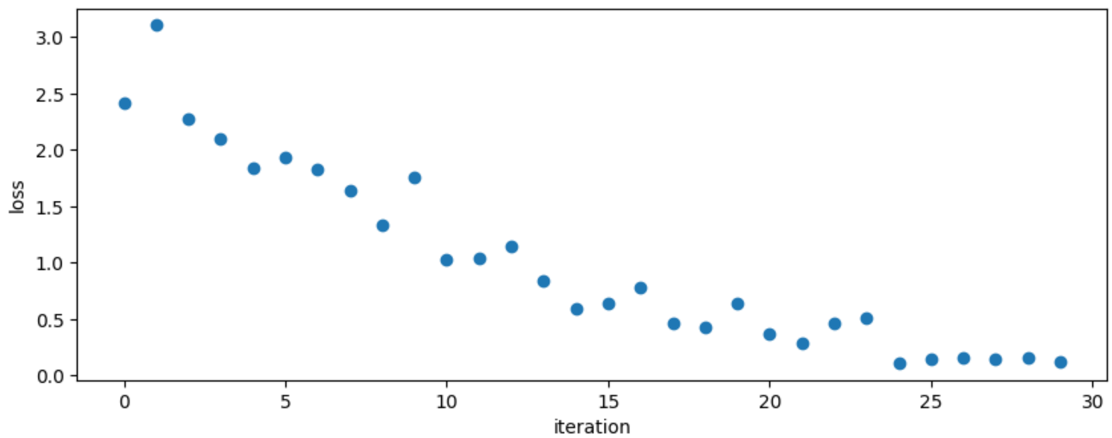
```
Small data training accuracy: 0.82
```

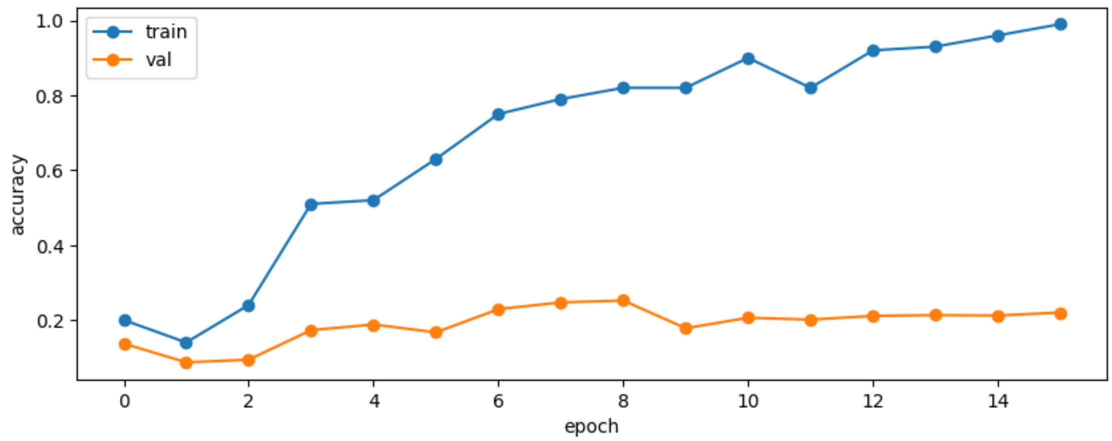
[19]:
```
# Print final validation accuracy.
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

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

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

[20]:
```
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:

[21]:
```
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
```

[22]:
```
# Print final training accuracy.
print(
    "Full data training accuracy:",
    solver.check_accuracy(data['X_train'], data['y_train'])
)
```

```
Full data training accuracy: 0.47618367346938778
```

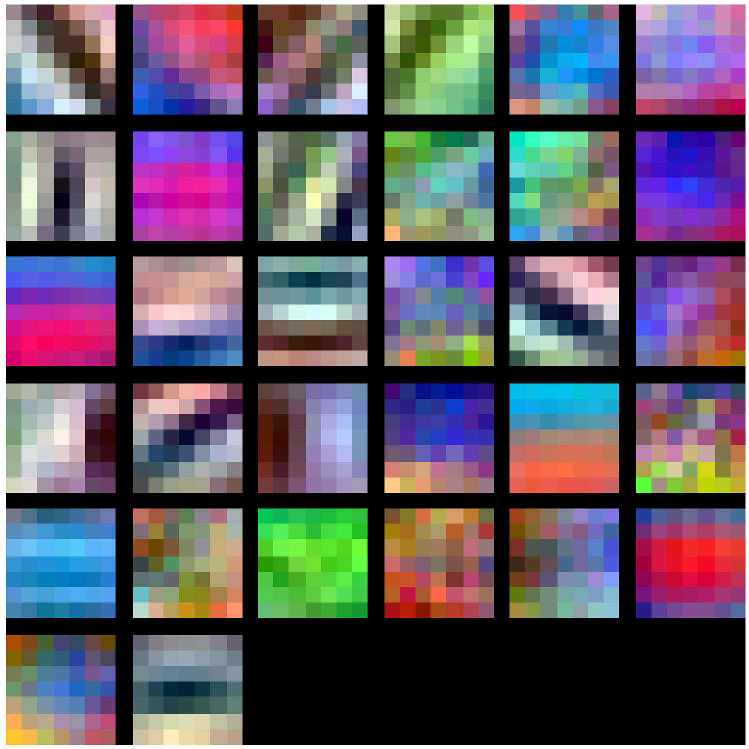
[23]:
```
# Print final validation accuracy.
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
```

```
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:

[24]:
```
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:

[26]:
```
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]
```

[27]:
```python
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:

[29]:
```python
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)
```

```python
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:  2.7866481948435e-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:

[30]:
```
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:

[38]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
```

```python
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('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:  7.413109384854475e-08
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12
```

# PyTorch

November 2, 2024

[ ]:
```
# 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.

[1]:
```
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: cuda
```

# 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.

[2]:
```
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).

[3]:
```python
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.

[4]:
```python
import torch.nn.functional as F  # useful stateless functions
```

```python
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?
    - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can␣
↪you
      figure out what the shape should be?

  Returns:
  - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
  """
  conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
  scores = None
 ␣
↪################################################################################
  # TODO: Implement the forward pass for the three-layer ConvNet.                ␣
↪  #
 ␣
↪################################################################################
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

  # *****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)
```

```python
# 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([[ 0.6845, -0.2657, -0.6812,  0.1488, -0.1727],
        [ 0.4862, -0.6444, -0.0280,  0.7609, -1.1040],
        [-1.2636, -1.3144, -1.1203,  0.4347,  0.3223]], device='cuda:0',
       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.

[15]:
```python
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.

[16]:
```python
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.

[17]:
```
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 = 3.4395
Checking accuracy on the val set
Got 152 / 1000 correct (15.20%)

Iteration 100, loss = 2.9898
Checking accuracy on the val set
Got 362 / 1000 correct (36.20%)

Iteration 200, loss = 2.0010
Checking accuracy on the val set
Got 379 / 1000 correct (37.90%)

Iteration 300, loss = 1.7141
Checking accuracy on the val set
Got 372 / 1000 correct (37.20%)

Iteration 400, loss = 1.9813
Checking accuracy on the val set
Got 445 / 1000 correct (44.50%)
```

```
Iteration 500, loss = 1.9032
Checking accuracy on the val set
Got 430 / 1000 correct (43.00%)

Iteration 600, loss = 1.7845
Checking accuracy on the val set
Got 454 / 1000 correct (45.40%)

Iteration 700, loss = 1.7163
Checking accuracy on the val set
Got 460 / 1000 correct (46.00%)
```

### 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.

[18]:
```
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)*****

# JIANTENG
```

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

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

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

```
Iteration 0, loss = 2.9493
Checking accuracy on the val set
Got 173 / 1000 correct (17.30%)

Iteration 100, loss = 1.6980
Checking accuracy on the val set
Got 308 / 1000 correct (30.80%)

Iteration 200, loss = 1.5482
Checking accuracy on the val set
Got 370 / 1000 correct (37.00%)

Iteration 300, loss = 1.7592
Checking accuracy on the val set
Got 387 / 1000 correct (38.70%)

Iteration 400, loss = 1.6685
Checking accuracy on the val set
Got 459 / 1000 correct (45.90%)

Iteration 500, loss = 1.5091
Checking accuracy on the val set
Got 456 / 1000 correct (45.60%)

Iteration 600, loss = 1.5317
Checking accuracy on the val set
Got 450 / 1000 correct (45.00%)

Iteration 700, loss = 1.3744
Checking accuracy on the val set
Got 476 / 1000 correct (47.60%)
```

# 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:

[19]:
```
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.

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

        # JIANTENG
        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1)
        self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)

        # *****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)*****

        # JIANTENG
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = flatten(x)
        scores = self.fc(x)

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


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

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

#### 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.

[21]:
```
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.

[22]:
```
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.

[23]:
```
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 = 3.9919
Checking accuracy on validation set
Got 122 / 1000 correct (12.20)

Iteration 100, loss = 2.2685
Checking accuracy on validation set
Got 336 / 1000 correct (33.60)

Iteration 200, loss = 2.0947
Checking accuracy on validation set
Got 337 / 1000 correct (33.70)

Iteration 300, loss = 2.0098
Checking accuracy on validation set
Got 372 / 1000 correct (37.20)

Iteration 400, loss = 1.9114
Checking accuracy on validation set
```

```
Got 426 / 1000 correct (42.60)

Iteration 500, loss = 2.2022
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Iteration 600, loss = 2.0270
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)

Iteration 700, loss = 1.9937
Checking accuracy on validation set
Got 445 / 1000 correct (44.50)
```

### 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.

[24]:
```
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)*****

# JIANTENG
model = ThreeLayerConvNet(3, channel_1, channel_2, 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 = 2.3435
Checking accuracy on validation set
Got 79 / 1000 correct (7.90)
```

```
Iteration 100, loss = 2.1306
Checking accuracy on validation set
Got 311 / 1000 correct (31.10)

Iteration 200, loss = 1.8211
Checking accuracy on validation set
Got 375 / 1000 correct (37.50)

Iteration 300, loss = 1.7798
Checking accuracy on validation set
Got 379 / 1000 correct (37.90)

Iteration 400, loss = 1.7136
Checking accuracy on validation set
Got 407 / 1000 correct (40.70)

Iteration 500, loss = 1.5992
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)

Iteration 600, loss = 1.6888
Checking accuracy on validation set
Got 445 / 1000 correct (44.50)

Iteration 700, loss = 1.6437
Checking accuracy on validation set
Got 452 / 1000 correct (45.20)
```

# 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.

[25]:
```python
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

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

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3313
Checking accuracy on validation set
Got 171 / 1000 correct (17.10)

Iteration 100, loss = 1.5929
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Iteration 200, loss = 1.9004
Checking accuracy on validation set
Got 390 / 1000 correct (39.00)

Iteration 300, loss = 1.5852
Checking accuracy on validation set
Got 440 / 1000 correct (44.00)

Iteration 400, loss = 1.9816
Checking accuracy on validation set
Got 401 / 1000 correct (40.10)

Iteration 500, loss = 1.2676
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 600, loss = 2.0081
Checking accuracy on validation set
```

```
Got 436 / 1000 correct (43.60)

Iteration 700, loss = 1.7204
Checking accuracy on validation set
Got 442 / 1000 correct (44.20)
```

### 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.

[28]:
```
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)*****

# JIANTENG
model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2 * 32 * 32, 10)
)
optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9,␣
 ↪nesterov=True)
```

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.2969
Checking accuracy on validation set
Got 143 / 1000 correct (14.30)

Iteration 100, loss = 1.9011
Checking accuracy on validation set
Got 447 / 1000 correct (44.70)

Iteration 200, loss = 1.3034
Checking accuracy on validation set
Got 476 / 1000 correct (47.60)

Iteration 300, loss = 1.1136
Checking accuracy on validation set
Got 515 / 1000 correct (51.50)

Iteration 400, loss = 1.6643
Checking accuracy on validation set
Got 532 / 1000 correct (53.20)

Iteration 500, loss = 1.4117
Checking accuracy on validation set
Got 569 / 1000 correct (56.90)

Iteration 600, loss = 1.3038
Checking accuracy on validation set
Got 565 / 1000 correct (56.50)

Iteration 700, loss = 1.2257
Checking accuracy on validation set
Got 549 / 1000 correct (54.90)
```

## 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!

[29]:
```
################################################################################
# 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)*****

# JIANTENG

# Define a deeper network architecture with batch normalization and dropout
class CustomConvNet(nn.Module):
    def __init__(self, in_channel, num_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channel, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(256 * 4 * 4, 1024)
        self.fc2 = nn.Linear(1024, num_classes)
        self.dropout = nn.Dropout(0.5)
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
```

```python
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = flatten(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Instantiate the model and optimizer
model = CustomConvNet(3, 10)
optimizer = optim.Adam(model.parameters(), lr=1e-3)


# *****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 = 2.4595
Checking accuracy on validation set
Got 178 / 1000 correct (17.80)

Iteration 100, loss = 1.6347
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Iteration 200, loss = 1.5809
Checking accuracy on validation set
Got 461 / 1000 correct (46.10)

Iteration 300, loss = 1.5851
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Iteration 400, loss = 1.4662
Checking accuracy on validation set
Got 514 / 1000 correct (51.40)

Iteration 500, loss = 1.3529
Checking accuracy on validation set
Got 551 / 1000 correct (55.10)

Iteration 600, loss = 1.6140
Checking accuracy on validation set
Got 573 / 1000 correct (57.30)
```

```
Iteration 700, loss = 1.1058
Checking accuracy on validation set
Got 619 / 1000 correct (61.90)

Iteration 0, loss = 1.3154
Checking accuracy on validation set
Got 615 / 1000 correct (61.50)

Iteration 100, loss = 1.2027
Checking accuracy on validation set
Got 595 / 1000 correct (59.50)

Iteration 200, loss = 1.0369
Checking accuracy on validation set
Got 656 / 1000 correct (65.60)

Iteration 300, loss = 1.2053
Checking accuracy on validation set
Got 659 / 1000 correct (65.90)

Iteration 400, loss = 1.0968
Checking accuracy on validation set
Got 666 / 1000 correct (66.60)

Iteration 500, loss = 0.9826
Checking accuracy on validation set
Got 673 / 1000 correct (67.30)

Iteration 600, loss = 1.1571
Checking accuracy on validation set
Got 686 / 1000 correct (68.60)

Iteration 700, loss = 0.9958
Checking accuracy on validation set
Got 665 / 1000 correct (66.50)

Iteration 0, loss = 1.0090
Checking accuracy on validation set
Got 701 / 1000 correct (70.10)

Iteration 100, loss = 0.6990
Checking accuracy on validation set
Got 706 / 1000 correct (70.60)

Iteration 200, loss = 1.0547
Checking accuracy on validation set
Got 699 / 1000 correct (69.90)
```

```
Iteration 300, loss = 0.7392
Checking accuracy on validation set
Got 692 / 1000 correct (69.20)

Iteration 400, loss = 0.8808
Checking accuracy on validation set
Got 664 / 1000 correct (66.40)

Iteration 500, loss = 0.9809
Checking accuracy on validation set
Got 728 / 1000 correct (72.80)

Iteration 600, loss = 0.6375
Checking accuracy on validation set
Got 719 / 1000 correct (71.90)

Iteration 700, loss = 0.6162
Checking accuracy on validation set
Got 732 / 1000 correct (73.20)

Iteration 0, loss = 0.8764
Checking accuracy on validation set
Got 734 / 1000 correct (73.40)

Iteration 100, loss = 0.6485
Checking accuracy on validation set
Got 687 / 1000 correct (68.70)

Iteration 200, loss = 0.9376
Checking accuracy on validation set
Got 725 / 1000 correct (72.50)

Iteration 300, loss = 0.7794
Checking accuracy on validation set
Got 736 / 1000 correct (73.60)

Iteration 400, loss = 0.6322
Checking accuracy on validation set
Got 739 / 1000 correct (73.90)

Iteration 500, loss = 0.7971
Checking accuracy on validation set
Got 770 / 1000 correct (77.00)

Iteration 600, loss = 0.8357
Checking accuracy on validation set
Got 759 / 1000 correct (75.90)
```

```
Iteration 700, loss = 0.8979
Checking accuracy on validation set
Got 751 / 1000 correct (75.10)

Iteration 0, loss = 0.5565
Checking accuracy on validation set
Got 766 / 1000 correct (76.60)

Iteration 100, loss = 0.7453
Checking accuracy on validation set
Got 766 / 1000 correct (76.60)

Iteration 200, loss = 0.6575
Checking accuracy on validation set
Got 758 / 1000 correct (75.80)

Iteration 300, loss = 0.6470
Checking accuracy on validation set
Got 768 / 1000 correct (76.80)

Iteration 400, loss = 0.9100
Checking accuracy on validation set
Got 719 / 1000 correct (71.90)

Iteration 500, loss = 0.5788
Checking accuracy on validation set
Got 760 / 1000 correct (76.00)

Iteration 600, loss = 1.1178
Checking accuracy on validation set
Got 729 / 1000 correct (72.90)

Iteration 700, loss = 0.6173
Checking accuracy on validation set
Got 774 / 1000 correct (77.40)

Iteration 0, loss = 0.5221
Checking accuracy on validation set
Got 774 / 1000 correct (77.40)

Iteration 100, loss = 0.5589
Checking accuracy on validation set
Got 780 / 1000 correct (78.00)

Iteration 200, loss = 0.6971
Checking accuracy on validation set
Got 733 / 1000 correct (73.30)
```

```
Iteration 300, loss = 0.4473
Checking accuracy on validation set
Got 779 / 1000 correct (77.90)

Iteration 400, loss = 0.4703
Checking accuracy on validation set
Got 766 / 1000 correct (76.60)

Iteration 500, loss = 0.6054
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 600, loss = 0.8164
Checking accuracy on validation set
Got 759 / 1000 correct (75.90)

Iteration 700, loss = 0.4836
Checking accuracy on validation set
Got 790 / 1000 correct (79.00)

Iteration 0, loss = 0.4443
Checking accuracy on validation set
Got 773 / 1000 correct (77.30)

Iteration 100, loss = 0.5736
Checking accuracy on validation set
Got 783 / 1000 correct (78.30)

Iteration 200, loss = 0.8475
Checking accuracy on validation set
Got 761 / 1000 correct (76.10)

Iteration 300, loss = 0.5076
Checking accuracy on validation set
Got 772 / 1000 correct (77.20)

Iteration 400, loss = 0.4316
Checking accuracy on validation set
Got 781 / 1000 correct (78.10)

Iteration 500, loss = 0.4393
Checking accuracy on validation set
Got 803 / 1000 correct (80.30)

Iteration 600, loss = 0.3263
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)
```

```
Iteration 700, loss = 0.5405
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)

Iteration 0, loss = 0.3379
Checking accuracy on validation set
Got 787 / 1000 correct (78.70)

Iteration 100, loss = 0.3491
Checking accuracy on validation set
Got 784 / 1000 correct (78.40)

Iteration 200, loss = 0.5830
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 300, loss = 0.3137
Checking accuracy on validation set
Got 798 / 1000 correct (79.80)

Iteration 400, loss = 0.4601
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 500, loss = 0.2637
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 600, loss = 0.4526
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 700, loss = 0.6879
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 0, loss = 0.2170
Checking accuracy on validation set
Got 789 / 1000 correct (78.90)

Iteration 100, loss = 0.3848
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 200, loss = 0.3279
Checking accuracy on validation set
Got 788 / 1000 correct (78.80)
```

```
Iteration 300, loss = 0.4966
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 400, loss = 0.3849
Checking accuracy on validation set
Got 820 / 1000 correct (82.00)

Iteration 500, loss = 0.4989
Checking accuracy on validation set
Got 820 / 1000 correct (82.00)

Iteration 600, loss = 0.3349
Checking accuracy on validation set
Got 828 / 1000 correct (82.80)

Iteration 700, loss = 0.5929
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)

Iteration 0, loss = 0.3059
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 100, loss = 0.3678
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 200, loss = 0.3794
Checking accuracy on validation set
Got 798 / 1000 correct (79.80)

Iteration 300, loss = 0.3057
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)

Iteration 400, loss = 0.2808
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 500, loss = 0.2452
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 600, loss = 0.6448
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)
```

```
Iteration 700, loss = 0.3504
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)
```

## 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:**

To tackle the CIFAR-10 open-ended challenge, I experimented with a custom CNN architecture that includes batch normalization and dropout for improved generalization and training stability. Here's a summary of the design choices and the rationale behind each:

1. **Network Architecture**:
    - The model consists of three convolutional layers followed by batch normalization and ReLU activations. Batch normalization helps normalize activations across the network, accelerating convergence and improving generalization.
    - Each convolutional layer uses a 3x3 kernel with padding, followed by a max-pooling layer to reduce the spatial dimensions. The use of max-pooling after each convolution allows the network to capture essential features while reducing computation.
    - After the convolutional layers, I flattened the output and added two fully connected layers. I included a dropout layer with a probability of 0.5 between these layers to reduce overfitting, ensuring the model doesn't rely heavily on any single feature during training.
2. **Optimizer**:
    - I used the Adam optimizer with a learning rate of $1e^{-3}$. Adam is adaptive, which helps in quickly converging to a good solution without extensive learning rate tuning.
3. **Hyperparameters**:
    - I trained the model for 10 epochs, monitoring the accuracy on the validation set. By testing with batch normalization, dropout, and Adam optimizer, the model achieved a validation accuracy of at least 70% within the specified epochs.
4. **Additional Features to Explore**:
    - Further enhancements could include using data augmentation to increase the robustness of the model, experimenting with other optimizers like RMSprop, or increasing the depth of the network for potentially higher accuracy.

In conclusion, the combination of batch normalization, dropout, and a deeper network structure enabled the model to achieve the required accuracy within the given constraints.

## 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.

[31]:

```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 8144 / 10000 correct (81.44)
```