FullyConnectedNets

November 5, 2022

[ ]:
```
# 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/assignment1/'
FOLDERNAME = 'cs231n/assignment2'
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
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignment2
```

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.

```
[ ]: # 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.
```

```
[ ]: # 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.

[ ]:
```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])}")
```

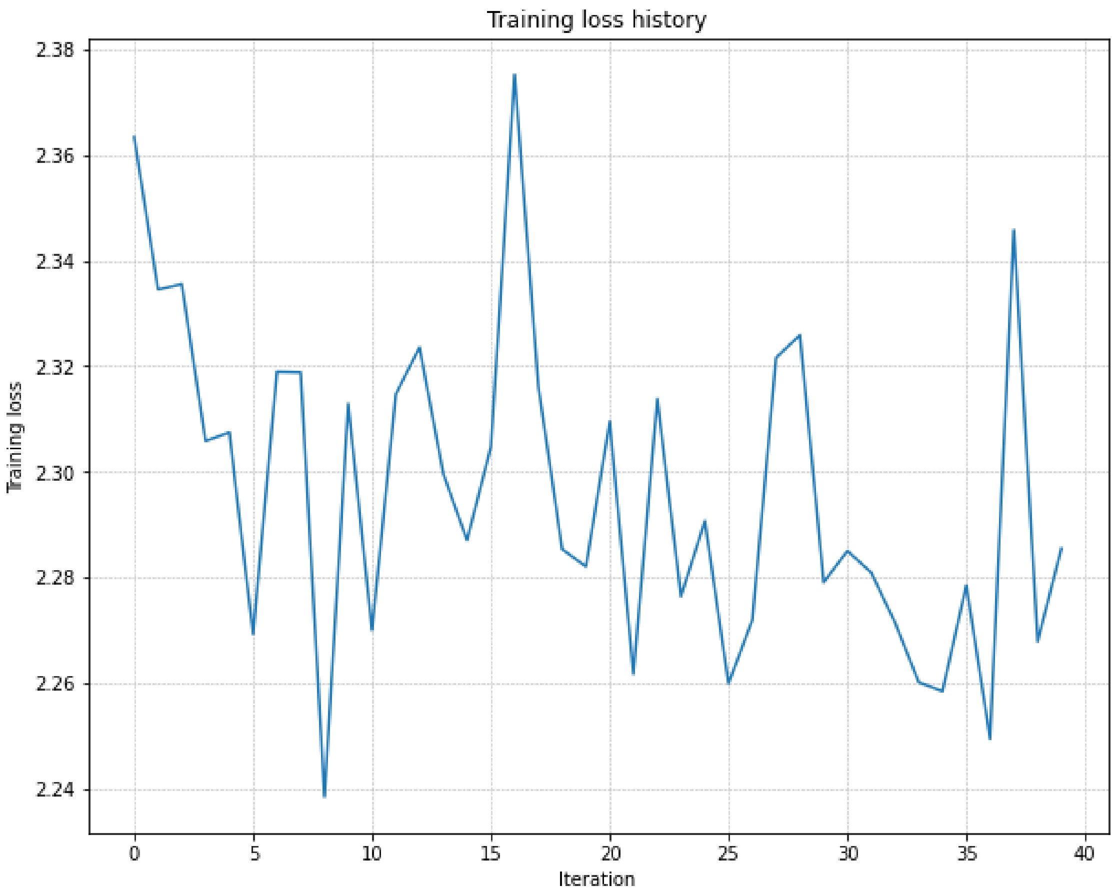
```
Running check with reg =  0
Initial loss:  2.300479089768492
W1 relative error: 1.0252674471656573e-07
W2 relative error: 2.2120479295080622e-05
W3 relative error: 4.5623278736665505e-07
b1 relative error: 1.0
b2 relative error: 1.0
b3 relative error: 1.0
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 3.904541941902138e-09
W2 relative error: 6.86942277940646e-08
W3 relative error: 3.483989247437803e-08
b1 relative error: 1.0
b2 relative error: 1.0
b3 relative error: 1.0
```

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.

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

     num_train = 50
     small_data = {
       "X_train": data["X_train"][:num_train],
       "y_train": data["y_train"][:num_train],
       "X_val": data["X_val"],
       "y_val": data["y_val"],
     }

     weight_scale = 1e-2   # Experiment with this!
     learning_rate = 1e-4  # Experiment with this!
     model = FullyConnectedNet(
         [100, 100],
         weight_scale=weight_scale,
         dtype=np.float64
     )
     solver = Solver(
         model,
         small_data,
         print_every=10,
         num_epochs=20,
         batch_size=25,
         update_rule="sgd",
         optim_config={"learning_rate": learning_rate},
     )
     solver.train()

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

```
(Iteration 1 / 40) loss: 2.363364
(Epoch 0 / 20) train acc: 0.020000; val_acc: 0.105000
(Epoch 1 / 20) train acc: 0.020000; val_acc: 0.106000
(Epoch 2 / 20) train acc: 0.020000; val_acc: 0.110000
(Epoch 3 / 20) train acc: 0.020000; val_acc: 0.110000
(Epoch 4 / 20) train acc: 0.040000; val_acc: 0.109000
(Epoch 5 / 20) train acc: 0.040000; val_acc: 0.111000
(Iteration 11 / 40) loss: 2.270051
(Epoch 6 / 20) train acc: 0.040000; val_acc: 0.111000
(Epoch 7 / 20) train acc: 0.060000; val_acc: 0.112000
```

```
(Epoch 8 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 9 / 20) train acc: 0.040000; val_acc: 0.110000
(Epoch 10 / 20) train acc: 0.060000; val_acc: 0.109000
(Iteration 21 / 40) loss: 2.309581
(Epoch 11 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 12 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 13 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 14 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 15 / 20) train acc: 0.060000; val_acc: 0.113000
(Iteration 31 / 40) loss: 2.285072
(Epoch 16 / 20) train acc: 0.060000; val_acc: 0.117000
(Epoch 17 / 20) train acc: 0.080000; val_acc: 0.113000
(Epoch 18 / 20) train acc: 0.080000; val_acc: 0.118000
(Epoch 19 / 20) train acc: 0.100000; val_acc: 0.118000
(Epoch 20 / 20) train acc: 0.100000; val_acc: 0.120000
```



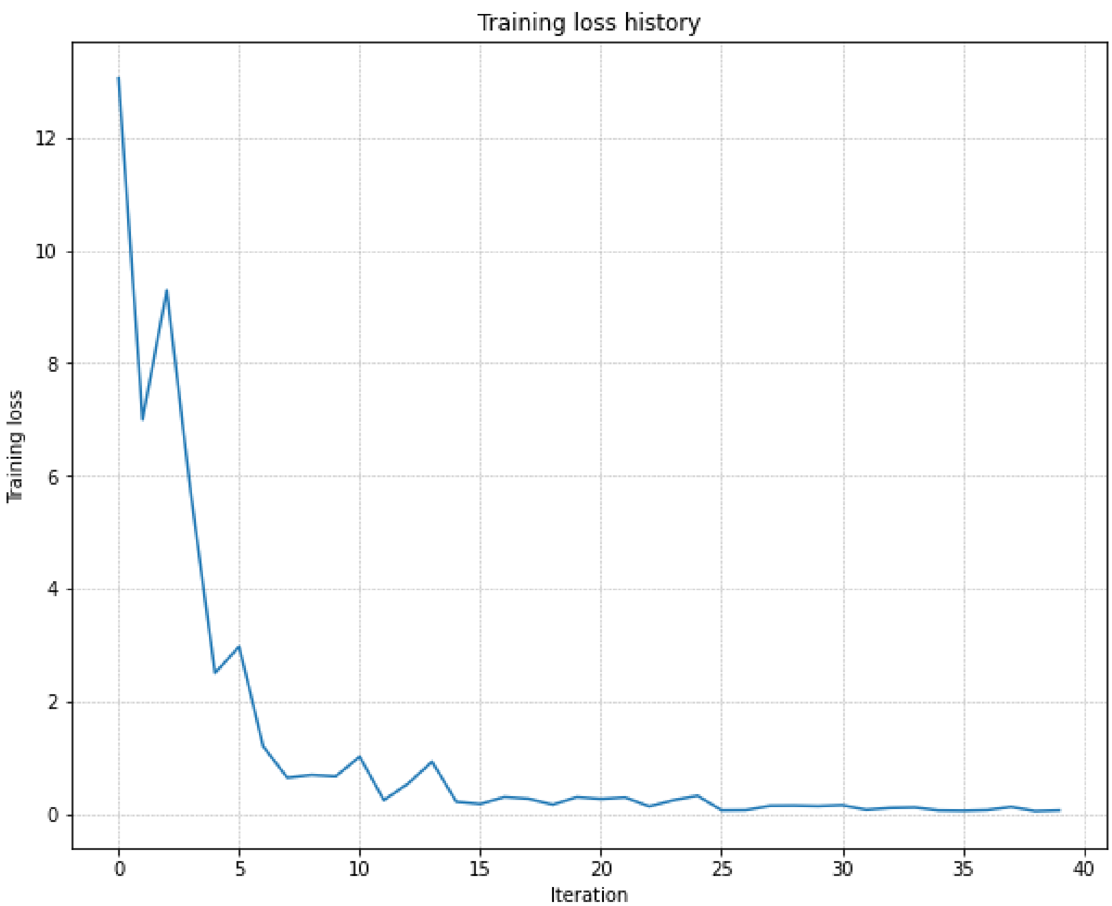
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[ ]:

```python
# 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 = 6e-2   # Experiment with this!
#Tried Lr[ 1e-1(vanishing gradients),2e-2,2e-3,2e-4,2e-5(exploding)]
#Experimented WS= [1e-1,2e-2,3e-3,4e-4,3e-2,4e-2,5e-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: 13.054907
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.115000
(Epoch 1 / 20) train acc: 0.240000; val_acc: 0.088000
(Epoch 2 / 20) train acc: 0.340000; val_acc: 0.136000
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.132000
(Epoch 4 / 20) train acc: 0.740000; val_acc: 0.131000
(Epoch 5 / 20) train acc: 0.840000; val_acc: 0.127000
(Iteration 11 / 40) loss: 1.024604
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.133000
```

```
(Epoch 7 / 20) train acc: 0.880000; val_acc: 0.132000
(Epoch 8 / 20) train acc: 0.900000; val_acc: 0.127000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.126000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.121000
(Iteration 21 / 40) loss: 0.274109
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.127000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.132000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.132000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.128000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.128000
(Iteration 31 / 40) loss: 0.163140
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.130000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.134000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.132000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.128000
```



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:

5 layer network was much more sensitive to the initialization scale.the sensitivity is due to the fact of number of layers is higher in 5 layer NN. Greater number of NN, the small gradients tends to zero whereas higher gradietns tend to 1(sigmoid).

Asssigning lower ws values leads to exploding gradient, whereas higher ws values leads to vanishing. Thus optmising hyperparams is difficult for 5 layer NN. [Mentioned all tried values]

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.

[ ]:
```
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
```

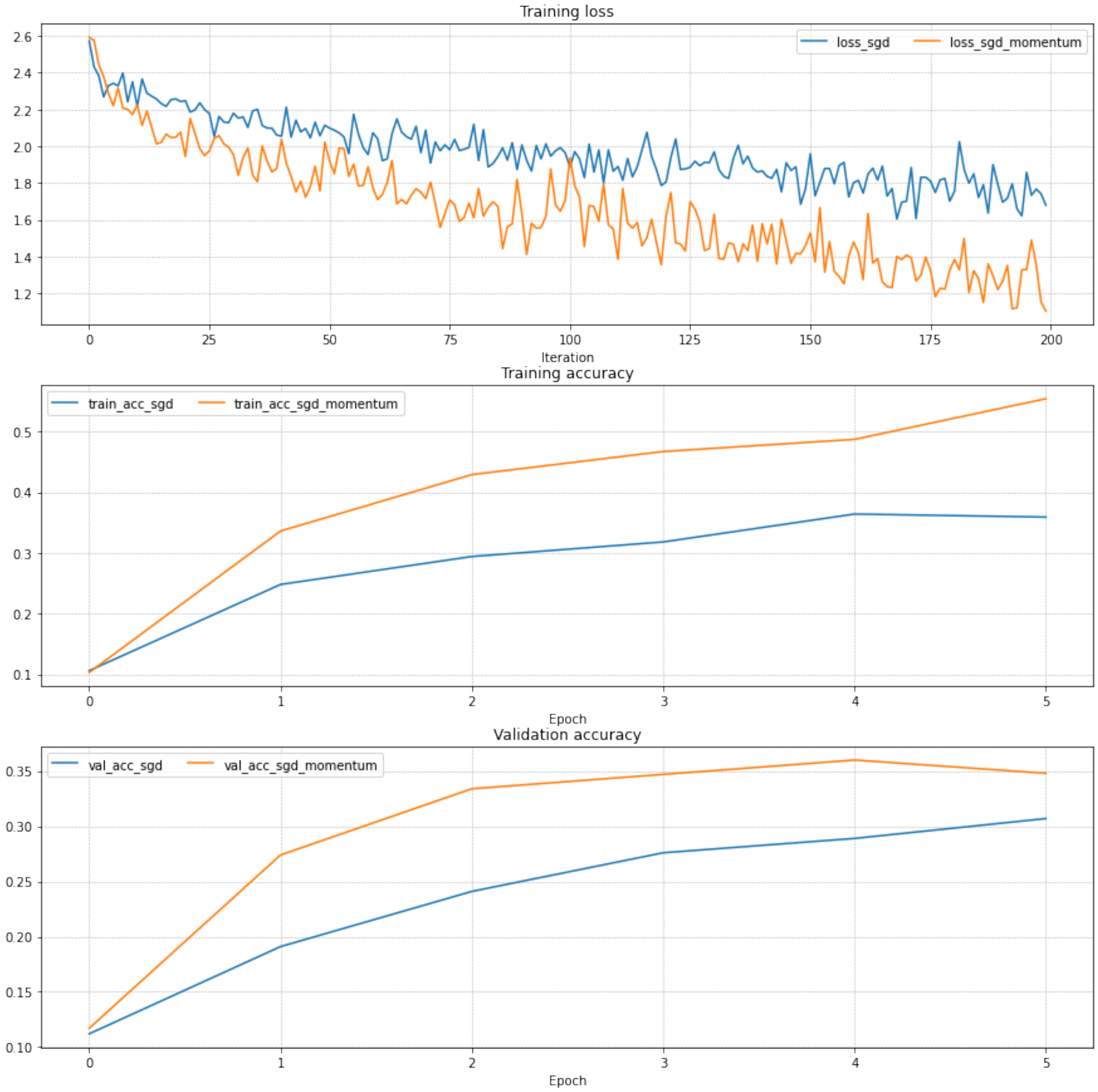
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[ ]:
```
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')
```

```python
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: 2.573687
(Epoch 0 / 5) train acc: 0.106000; val_acc: 0.112000
(Iteration 11 / 200) loss: 2.217090
(Iteration 21 / 200) loss: 2.247937
(Iteration 31 / 200) loss: 2.179834
(Epoch 1 / 5) train acc: 0.248000; val_acc: 0.191000
(Iteration 41 / 200) loss: 2.054746
(Iteration 51 / 200) loss: 2.098385
(Iteration 61 / 200) loss: 2.040300
(Iteration 71 / 200) loss: 2.087600
(Epoch 2 / 5) train acc: 0.294000; val_acc: 0.241000
(Iteration 81 / 200) loss: 2.119723
(Iteration 91 / 200) loss: 2.007485
(Iteration 101 / 200) loss: 1.887069
(Iteration 111 / 200) loss: 1.889993
(Epoch 3 / 5) train acc: 0.318000; val_acc: 0.276000
(Iteration 121 / 200) loss: 1.806010
(Iteration 131 / 200) loss: 1.969500
(Iteration 141 / 200) loss: 1.866526
(Iteration 151 / 200) loss: 1.958818
(Epoch 4 / 5) train acc: 0.364000; val_acc: 0.289000
(Iteration 161 / 200) loss: 1.814701
(Iteration 171 / 200) loss: 1.700205
(Iteration 181 / 200) loss: 1.755382
(Iteration 191 / 200) loss: 1.695881
(Epoch 5 / 5) train acc: 0.359000; val_acc: 0.307000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 2.592141
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.117000
```

```
(Iteration 11 / 200) loss: 2.224817
(Iteration 21 / 200) loss: 1.944476
(Iteration 31 / 200) loss: 1.954180
(Epoch 1 / 5) train acc: 0.336000; val_acc: 0.274000
(Iteration 41 / 200) loss: 2.038167
(Iteration 51 / 200) loss: 1.920163
(Iteration 61 / 200) loss: 1.710173
(Iteration 71 / 200) loss: 1.715595
(Epoch 2 / 5) train acc: 0.429000; val_acc: 0.334000
(Iteration 81 / 200) loss: 1.610300
(Iteration 91 / 200) loss: 1.632607
(Iteration 101 / 200) loss: 1.936593
(Iteration 111 / 200) loss: 1.384685
(Epoch 3 / 5) train acc: 0.467000; val_acc: 0.347000
(Iteration 121 / 200) loss: 1.621032
(Iteration 131 / 200) loss: 1.630881
(Iteration 141 / 200) loss: 1.578809
(Iteration 151 / 200) loss: 1.529214
(Epoch 4 / 5) train acc: 0.487000; val_acc: 0.360000
(Iteration 161 / 200) loss: 1.421826
(Iteration 171 / 200) loss: 1.407846
(Iteration 181 / 200) loss: 1.384722
(Iteration 191 / 200) loss: 1.264966
(Epoch 5 / 5) train acc: 0.554000; val_acc: 0.348000
```



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.

[ ]:
```
# 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
```

[ ]:
```
# 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
```

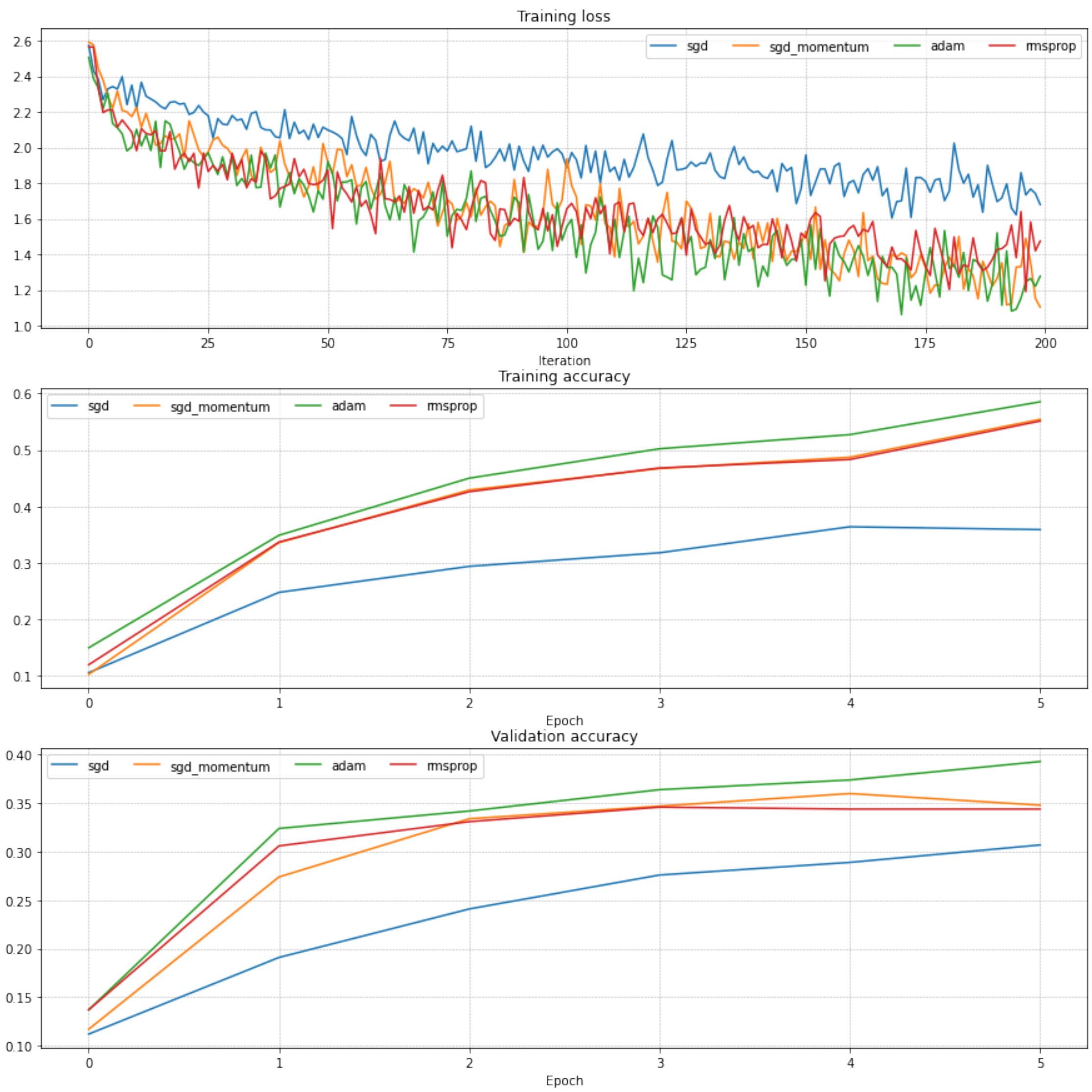
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[ ]:
```
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')
```

```python
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: 2.505630
(Epoch 0 / 5) train acc: 0.150000; val_acc: 0.137000
(Iteration 11 / 200) loss: 2.102257
(Iteration 21 / 200) loss: 1.877587
(Iteration 31 / 200) loss: 1.953238
(Epoch 1 / 5) train acc: 0.349000; val_acc: 0.324000
(Iteration 41 / 200) loss: 1.665255
(Iteration 51 / 200) loss: 1.921013
(Iteration 61 / 200) loss: 1.750838
(Iteration 71 / 200) loss: 1.610503
(Epoch 2 / 5) train acc: 0.450000; val_acc: 0.342000
(Iteration 81 / 200) loss: 1.868253
(Iteration 91 / 200) loss: 1.688751
(Iteration 101 / 200) loss: 1.555795
(Iteration 111 / 200) loss: 1.597575
(Epoch 3 / 5) train acc: 0.502000; val_acc: 0.364000
(Iteration 121 / 200) loss: 1.283666
(Iteration 131 / 200) loss: 1.483321
(Iteration 141 / 200) loss: 1.217880
(Iteration 151 / 200) loss: 1.225684
(Epoch 4 / 5) train acc: 0.527000; val_acc: 0.374000
(Iteration 161 / 200) loss: 1.382436
(Iteration 171 / 200) loss: 1.060959
(Iteration 181 / 200) loss: 1.320742
(Iteration 191 / 200) loss: 1.520575
(Epoch 5 / 5) train acc: 0.585000; val_acc: 0.393000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.565737
(Epoch 0 / 5) train acc: 0.120000; val_acc: 0.137000
(Iteration 11 / 200) loss: 1.982296
```

```
(Iteration 21 / 200) loss: 1.967813
(Iteration 31 / 200) loss: 1.968259
(Epoch 1 / 5) train acc: 0.337000; val_acc: 0.306000
(Iteration 41 / 200) loss: 1.769684
(Iteration 51 / 200) loss: 1.873704
(Iteration 61 / 200) loss: 1.515650
(Iteration 71 / 200) loss: 1.648068
(Epoch 2 / 5) train acc: 0.426000; val_acc: 0.331000
(Iteration 81 / 200) loss: 1.659920
(Iteration 91 / 200) loss: 1.586325
(Iteration 101 / 200) loss: 1.656131
(Iteration 111 / 200) loss: 1.678848
(Epoch 3 / 5) train acc: 0.468000; val_acc: 0.346000
(Iteration 121 / 200) loss: 1.625316
(Iteration 131 / 200) loss: 1.470133
(Iteration 141 / 200) loss: 1.436190
(Iteration 151 / 200) loss: 1.491996
(Epoch 4 / 5) train acc: 0.483000; val_acc: 0.344000
(Iteration 161 / 200) loss: 1.562179
(Iteration 171 / 200) loss: 1.374758
(Iteration 181 / 200) loss: 1.202025
(Iteration 191 / 200) loss: 1.427261
(Epoch 5 / 5) train acc: 0.551000; val_acc: 0.344000
```



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:

Once the learning rates are becoming smaller, the sum of gradients will never shrink making the model will stop learning early and move very slowly towards the local minimum.

In case of Adam, the constant addition momentum to the Adagrad framework enables it to tackle the decay rate problem.

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.

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

""" USed same values as two layer net with tweaks in learning rates"""

#we know the many hyperprams that can affect the accuracy are learning rate␣
 ↪alpha , weight_scale, regularization , input dimension, number of hidden␣
 ↪layers , batch size and number of epochs.
#Out of these many hyperparams we custom tune our model to get a best fit for␣
 ↪training data
learning_rates = [1e-2, 4e-3,5e-4]
#Used [1e-4,2e-4,3e-4] for experimenting
reg=[2.5e-3]
#Used for testing reg[5e-2,1e-3]
best_val = -1   # The highest validation accuracy that we have seen so far.
results = {}
#Create a empty dictionary to store accuracy values (same as we did in SVM and␣
 ↪softmax)
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
#We are given 2 tuning hyperparameters Lr and reg. In previous case, we used␣
↪hidden_dim as 50, now increasing it to 100. We define these two in loops for␣
↪the list of
#values given
for i in learning_rates:
  for j in reg:
    #Here we call the classifier and define values for train method
    model=FullyConnectedNet([50,50],reg=j)
    solver = Solver(model, data, update_rule='adam',␣
↪optim_config={'learning_rate': i},
                            lr_decay=0.95, num_epochs=10, batch_size=200,
                            print_every=100, verbose=True)
    solver.train()
        # store the best validation accuracy in model variable for the two␣
↪layer net by comparing with pre-given best_val
    if solver.best_val_acc > best_val:
        best_val = solver.best_val_acc
        best_model = model
    print('Validation accuracy: %.4f' % (solver.best_val_acc,))

print('Highest validation accuracy at:',best_val)


pass

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

```
(Iteration 1 / 2450) loss: 2.319597
(Epoch 0 / 10) train acc: 0.109000; val_acc: 0.102000
(Iteration 101 / 2450) loss: 2.430432
(Iteration 201 / 2450) loss: 2.329625
(Epoch 1 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 301 / 2450) loss: 2.370186
(Iteration 401 / 2450) loss: 2.312238
(Epoch 2 / 10) train acc: 0.087000; val_acc: 0.087000
(Iteration 501 / 2450) loss: 2.310475
(Iteration 601 / 2450) loss: 2.309827
(Iteration 701 / 2450) loss: 2.309295
(Epoch 3 / 10) train acc: 0.077000; val_acc: 0.087000
(Iteration 801 / 2450) loss: 2.308828
(Iteration 901 / 2450) loss: 2.308402
(Epoch 4 / 10) train acc: 0.115000; val_acc: 0.087000
(Iteration 1001 / 2450) loss: 2.308007
(Iteration 1101 / 2450) loss: 2.307649
```

```
(Iteration 1201 / 2450) loss: 2.307310
(Epoch 5 / 10) train acc: 0.089000; val_acc: 0.087000
(Iteration 1301 / 2450) loss: 2.307001
(Iteration 1401 / 2450) loss: 2.306711
(Epoch 6 / 10) train acc: 0.105000; val_acc: 0.087000
(Iteration 1501 / 2450) loss: 2.306439
(Iteration 1601 / 2450) loss: 2.306189
(Iteration 1701 / 2450) loss: 2.305951
(Epoch 7 / 10) train acc: 0.093000; val_acc: 0.087000
(Iteration 1801 / 2450) loss: 2.305734
(Iteration 1901 / 2450) loss: 2.305528
(Epoch 8 / 10) train acc: 0.091000; val_acc: 0.087000
(Iteration 2001 / 2450) loss: 2.305336
(Iteration 2101 / 2450) loss: 2.305159
(Iteration 2201 / 2450) loss: 2.304990
(Epoch 9 / 10) train acc: 0.095000; val_acc: 0.087000
(Iteration 2301 / 2450) loss: 2.304836
(Iteration 2401 / 2450) loss: 2.304690
(Epoch 10 / 10) train acc: 0.083000; val_acc: 0.087000
Validation accuracy: 0.1020
(Iteration 1 / 2450) loss: 2.329482
(Epoch 0 / 10) train acc: 0.127000; val_acc: 0.115000
(Iteration 101 / 2450) loss: 1.975455
(Iteration 201 / 2450) loss: 1.927924
(Epoch 1 / 10) train acc: 0.304000; val_acc: 0.318000
(Iteration 301 / 2450) loss: 1.864130
(Iteration 401 / 2450) loss: 1.913475
(Epoch 2 / 10) train acc: 0.335000; val_acc: 0.362000
(Iteration 501 / 2450) loss: 1.940394
(Iteration 601 / 2450) loss: 1.994142
(Iteration 701 / 2450) loss: 1.910728
(Epoch 3 / 10) train acc: 0.352000; val_acc: 0.351000
(Iteration 801 / 2450) loss: 1.818450
(Iteration 901 / 2450) loss: 1.817695
(Epoch 4 / 10) train acc: 0.366000; val_acc: 0.359000
(Iteration 1001 / 2450) loss: 2.047089
(Iteration 1101 / 2450) loss: 1.665030
(Iteration 1201 / 2450) loss: 1.710831
(Epoch 5 / 10) train acc: 0.388000; val_acc: 0.383000
(Iteration 1301 / 2450) loss: 1.845769
(Iteration 1401 / 2450) loss: 1.892521
(Epoch 6 / 10) train acc: 0.417000; val_acc: 0.376000
(Iteration 1501 / 2450) loss: 1.749617
(Iteration 1601 / 2450) loss: 1.825279
(Iteration 1701 / 2450) loss: 1.703739
(Epoch 7 / 10) train acc: 0.379000; val_acc: 0.392000
(Iteration 1801 / 2450) loss: 1.559110
(Iteration 1901 / 2450) loss: 1.760608
```

```
(Epoch 8 / 10) train acc: 0.405000; val_acc: 0.412000
(Iteration 2001 / 2450) loss: 1.837115
(Iteration 2101 / 2450) loss: 1.722555
(Iteration 2201 / 2450) loss: 1.687543
(Epoch 9 / 10) train acc: 0.422000; val_acc: 0.400000
(Iteration 2301 / 2450) loss: 1.699191
(Iteration 2401 / 2450) loss: 1.652903
(Epoch 10 / 10) train acc: 0.408000; val_acc: 0.388000
Validation accuracy: 0.4120
(Iteration 1 / 2450) loss: 2.329784
(Epoch 0 / 10) train acc: 0.151000; val_acc: 0.168000
(Iteration 101 / 2450) loss: 1.734288
(Iteration 201 / 2450) loss: 1.480371
(Epoch 1 / 10) train acc: 0.460000; val_acc: 0.452000
(Iteration 301 / 2450) loss: 1.460858
(Iteration 401 / 2450) loss: 1.459596
(Epoch 2 / 10) train acc: 0.507000; val_acc: 0.477000
(Iteration 501 / 2450) loss: 1.586606
(Iteration 601 / 2450) loss: 1.348510
(Iteration 701 / 2450) loss: 1.488565
(Epoch 3 / 10) train acc: 0.540000; val_acc: 0.481000
(Iteration 801 / 2450) loss: 1.366170
(Iteration 901 / 2450) loss: 1.415878
(Epoch 4 / 10) train acc: 0.515000; val_acc: 0.489000
(Iteration 1001 / 2450) loss: 1.318140
(Iteration 1101 / 2450) loss: 1.377277
(Iteration 1201 / 2450) loss: 1.396732
(Epoch 5 / 10) train acc: 0.539000; val_acc: 0.504000
(Iteration 1301 / 2450) loss: 1.467353
(Iteration 1401 / 2450) loss: 1.214609
(Epoch 6 / 10) train acc: 0.540000; val_acc: 0.498000
(Iteration 1501 / 2450) loss: 1.321887
(Iteration 1601 / 2450) loss: 1.289911
(Iteration 1701 / 2450) loss: 1.251740
(Epoch 7 / 10) train acc: 0.573000; val_acc: 0.510000
(Iteration 1801 / 2450) loss: 1.357002
(Iteration 1901 / 2450) loss: 1.412753
(Epoch 8 / 10) train acc: 0.568000; val_acc: 0.524000
(Iteration 2001 / 2450) loss: 1.184377
(Iteration 2101 / 2450) loss: 1.332263
(Iteration 2201 / 2450) loss: 1.215031
(Epoch 9 / 10) train acc: 0.570000; val_acc: 0.512000
(Iteration 2301 / 2450) loss: 1.112995
(Iteration 2401 / 2450) loss: 1.303280
(Epoch 10 / 10) train acc: 0.581000; val_acc: 0.527000
Validation accuracy: 0.5270
Highest validation accuracy at: 0.527
```

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.

[ ]:
```
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:  0.527
Test set accuracy:  0.517
```

BatchNormalization

November 5, 2022

[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/assignment1/'
FOLDERNAME = 'cs231n/assignment2'
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
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignment2
```

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.](https://arxiv.org/abs/1502.03167)

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

```
=========== 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.
```

[3]:
```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,)
```

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!

[ ]:
```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]
```

[ ]:
```
# 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.

[ ]:
```
# 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:  1.7029241291468676e-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 $X = \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_N \end{bmatrix}$,

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

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

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

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

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

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

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

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

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

bn_param = {'mode': 'train'}
out, cache = batchnorm_forward(x, gamma, beta, bn_param)

t1 = time.time()
dx1, dgamma1, dbeta1 = batchnorm_backward(dout, cache)
t2 = time.time()
dx2, dgamma2, dbeta2 = batchnorm_backward_alt(dout, cache)
t3 = time.time()

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

```
dx difference:  1.2661271829168436e-12
dgamma difference:  1.0
dbeta difference:  0.0
speedup: 2.95x
```

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`.

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

# You should expect losses between 1e-4~1e-10 for W,
# losses between 1e-08~1e-10 for b,
```

```
# and losses between 1e-08~1e-09 for beta and gammas.
for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64,
                            normalization='batchnorm')

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

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

```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 5.65e-06
W3 relative error: 4.14e-10
b1 relative error: 5.55e-08
b2 relative error: 2.98e-08
b3 relative error: 1.00e+00
beta1 relative error: 7.33e-09
beta2 relative error: 1.17e-09
gamma1 relative error: 7.47e-09
gamma2 relative error: 3.35e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 1.11e-08
b1 relative error: 4.16e-09
b2 relative error: 3.17e-08
b3 relative error: 1.00e+00
beta1 relative error: 6.65e-09
beta2 relative error: 3.48e-09
gamma1 relative error: 6.27e-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.

[23]:
```
np.random.seed(231)

# Try training a very deep net with batchnorm.
hidden_dims = [100, 100, 100, 100, 100]

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

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

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

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

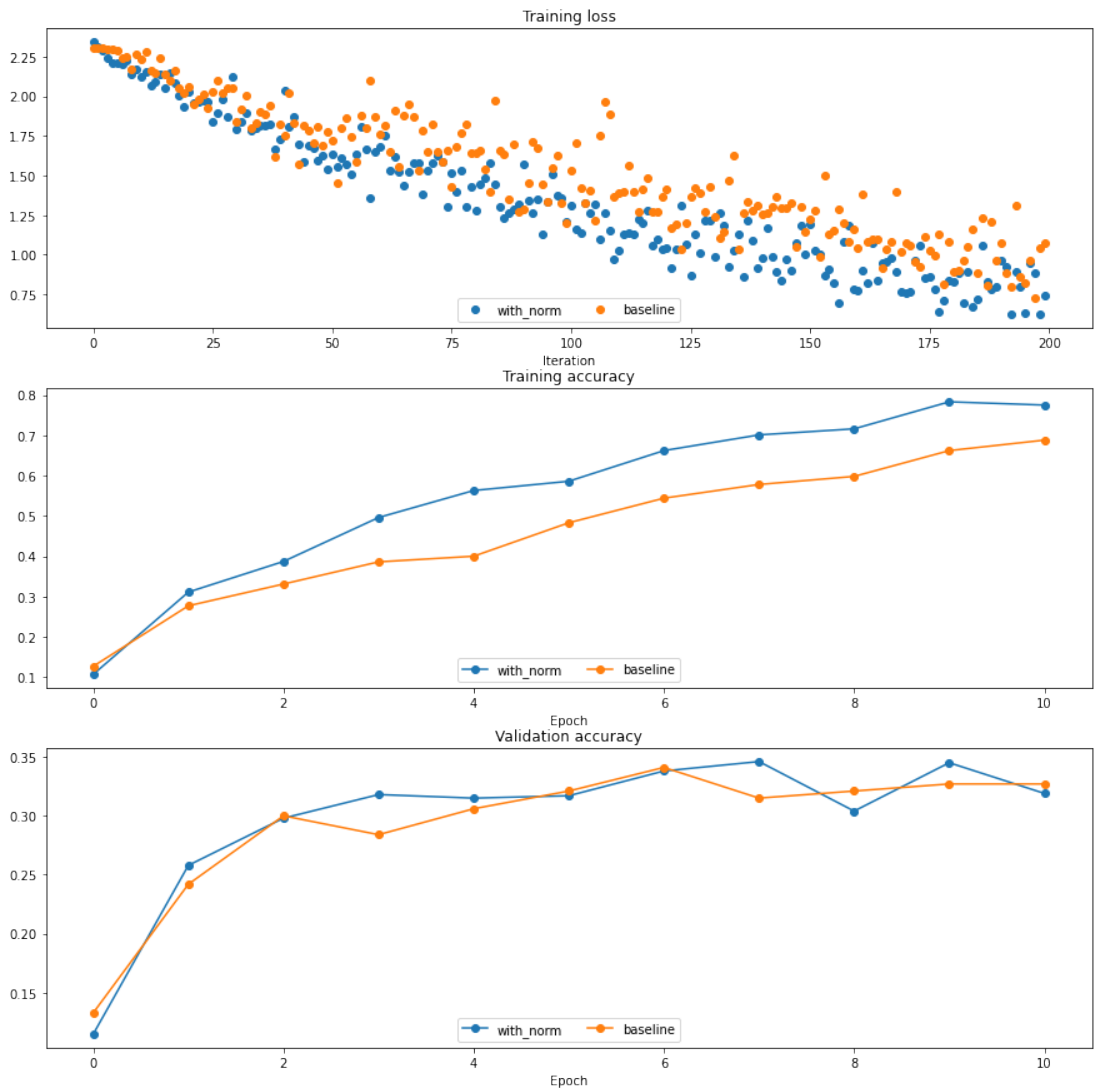
```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.311000; val_acc: 0.258000
```

```
(Iteration 21 / 200) loss: 2.027548
(Epoch 2 / 10) train acc: 0.387000; val_acc: 0.298000
(Iteration 41 / 200) loss: 2.032407
(Epoch 3 / 10) train acc: 0.496000; val_acc: 0.318000
(Iteration 61 / 200) loss: 1.678453
(Epoch 4 / 10) train acc: 0.563000; val_acc: 0.315000
(Iteration 81 / 200) loss: 1.282754
(Epoch 5 / 10) train acc: 0.586000; val_acc: 0.317000
(Iteration 101 / 200) loss: 1.308403
(Epoch 6 / 10) train acc: 0.662000; val_acc: 0.338000
(Iteration 121 / 200) loss: 1.043814
(Epoch 7 / 10) train acc: 0.701000; val_acc: 0.346000
(Iteration 141 / 200) loss: 0.982917
(Epoch 8 / 10) train acc: 0.716000; val_acc: 0.304000
(Iteration 161 / 200) loss: 0.772503
(Epoch 9 / 10) train acc: 0.783000; val_acc: 0.345000
(Iteration 181 / 200) loss: 0.828160
(Epoch 10 / 10) train acc: 0.775000; val_acc: 0.319000

Solver without batch norm:
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.127000; val_acc: 0.133000
(Epoch 1 / 10) train acc: 0.277000; val_acc: 0.242000
(Iteration 21 / 200) loss: 2.061509
(Epoch 2 / 10) train acc: 0.331000; val_acc: 0.300000
(Iteration 41 / 200) loss: 1.752483
(Epoch 3 / 10) train acc: 0.386000; val_acc: 0.284000
(Iteration 61 / 200) loss: 1.762878
(Epoch 4 / 10) train acc: 0.400000; val_acc: 0.306000
(Iteration 81 / 200) loss: 1.645409
(Epoch 5 / 10) train acc: 0.483000; val_acc: 0.321000
(Iteration 101 / 200) loss: 1.530852
(Epoch 6 / 10) train acc: 0.544000; val_acc: 0.341000
(Iteration 121 / 200) loss: 1.410970
(Epoch 7 / 10) train acc: 0.578000; val_acc: 0.315000
(Iteration 141 / 200) loss: 1.257347
(Epoch 8 / 10) train acc: 0.598000; val_acc: 0.321000
(Iteration 161 / 200) loss: 1.040411
(Epoch 9 / 10) train acc: 0.662000; val_acc: 0.327000
(Iteration 181 / 200) loss: 0.890304
(Epoch 10 / 10) train acc: 0.688000; val_acc: 0.327000
```

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.

[24]:
```python
def plot_training_history(title, label, baseline, bn_solvers, plot_fn,
↪bl_marker='.', bn_marker='.', labels=None):
    """utility function for plotting training history"""
```

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


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

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



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.

[25]:
```
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
```

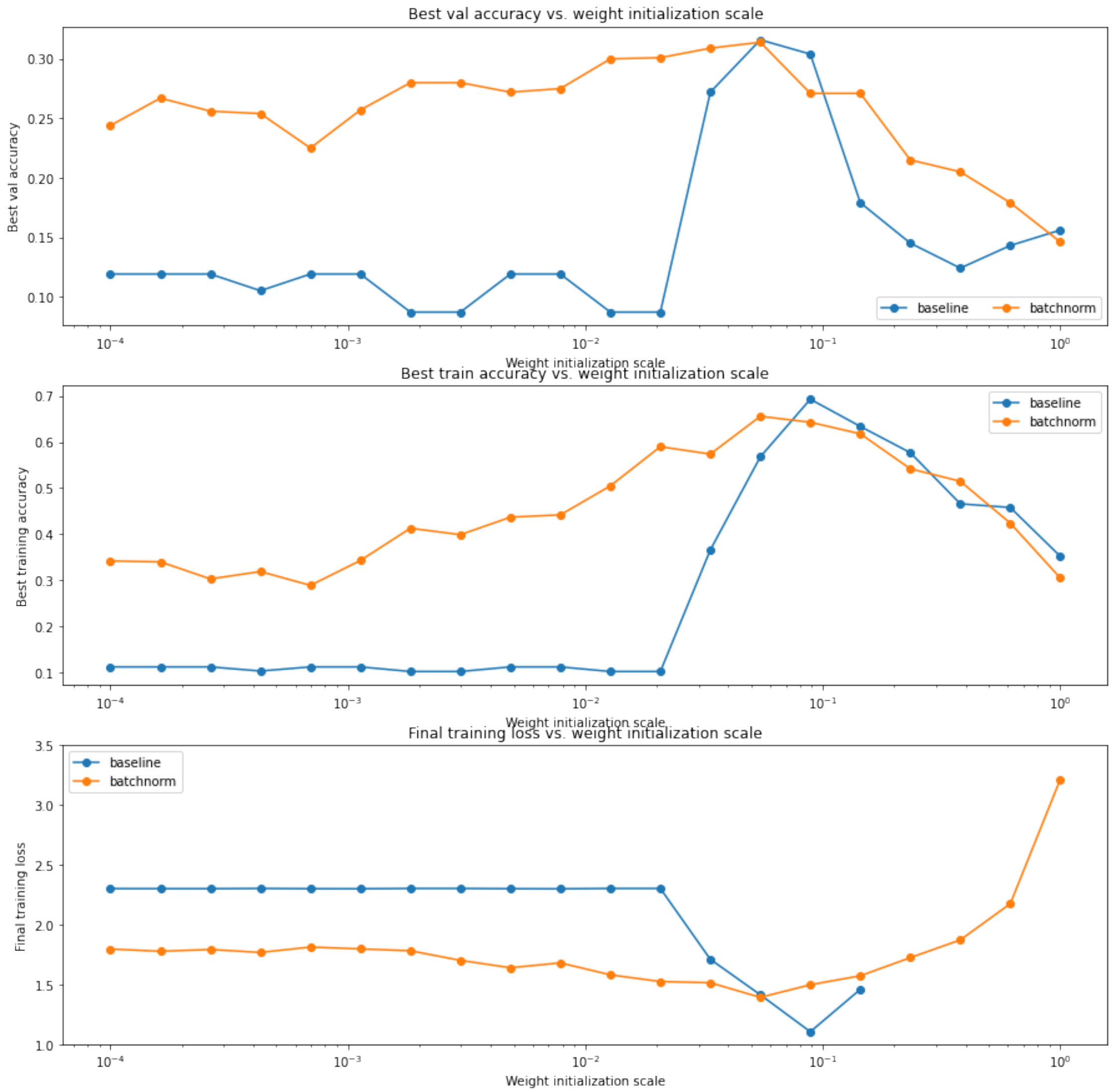
```
/content/drive/My Drive/cs231n/assignment2/cs231n/layers.py:185: RuntimeWarning:
divide by zero encountered in log
  loss=np.sum(-np.log(prb[np.arange(num_train), y]))
```

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

[ ]:

```
# 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()
```

```python
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:

Batchnorm improves the overall accuracy of the model. Eventhough there are exploding gradients visible in the third plot without batchnorm,the validation accuracy is similar.

Batchnorm reduces the occurance of vanishing gradients which are visible in first two plots.The batchnorm model is less sensitive to weight scales. The random assumption of weight scales makes the model perfrom poor, which we can see in graph 2.

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.

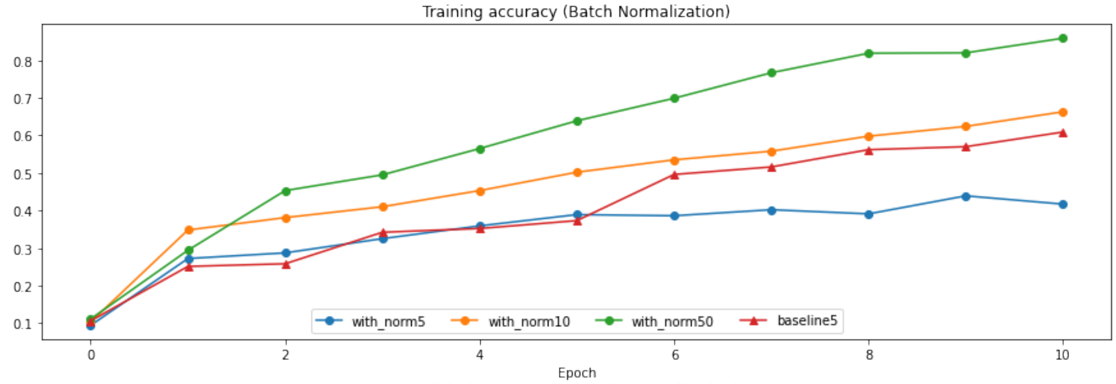
[10]:
```
def run_batchsize_experiments(normalization_mode):
    np.random.seed(231)

    # 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={
```

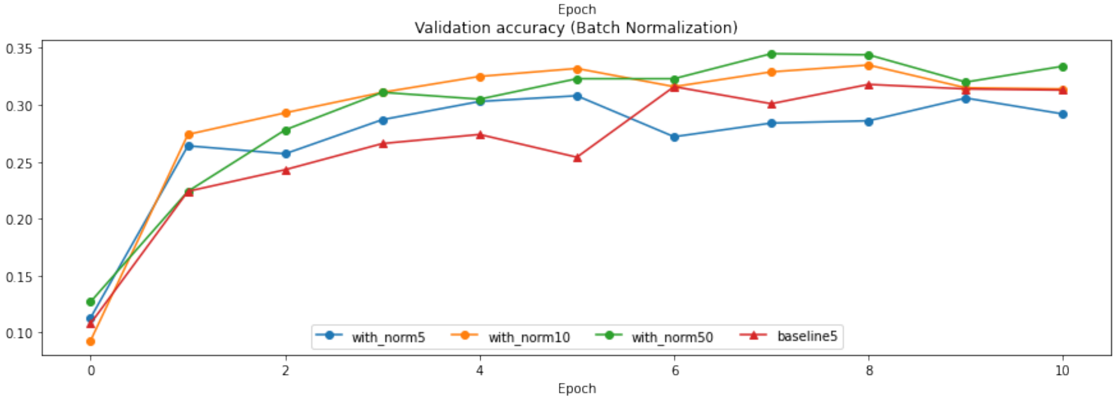
```
                     'learning_rate': lr,
                   },
                   verbose=False)
    solver.train()

    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
```

[ ]:

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

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





8.1 Inline Question 2:

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

8.2 Answer:

We can see that greater the batch size greater the accuracy in training and validation, whereas lesser the size, the lesser the accuracy than the baseline.

The model converges faster in fewer epochs in larger batch size, but iverall performance stabilizes after sometime as all model converges with other sizes.

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.](https://arxiv.org/pdf/1607.06450.pdf)

9.1 Inline Question 3:

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

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

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

9.2 Answer:

1 and 2 - Layer Norm as the scaling is done in each pixel of the vector row

3- BatchNorm as it is normalizes each feature through the image mini batch.

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.

[55]:
```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]
```

[54]:
```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)

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

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

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

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

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

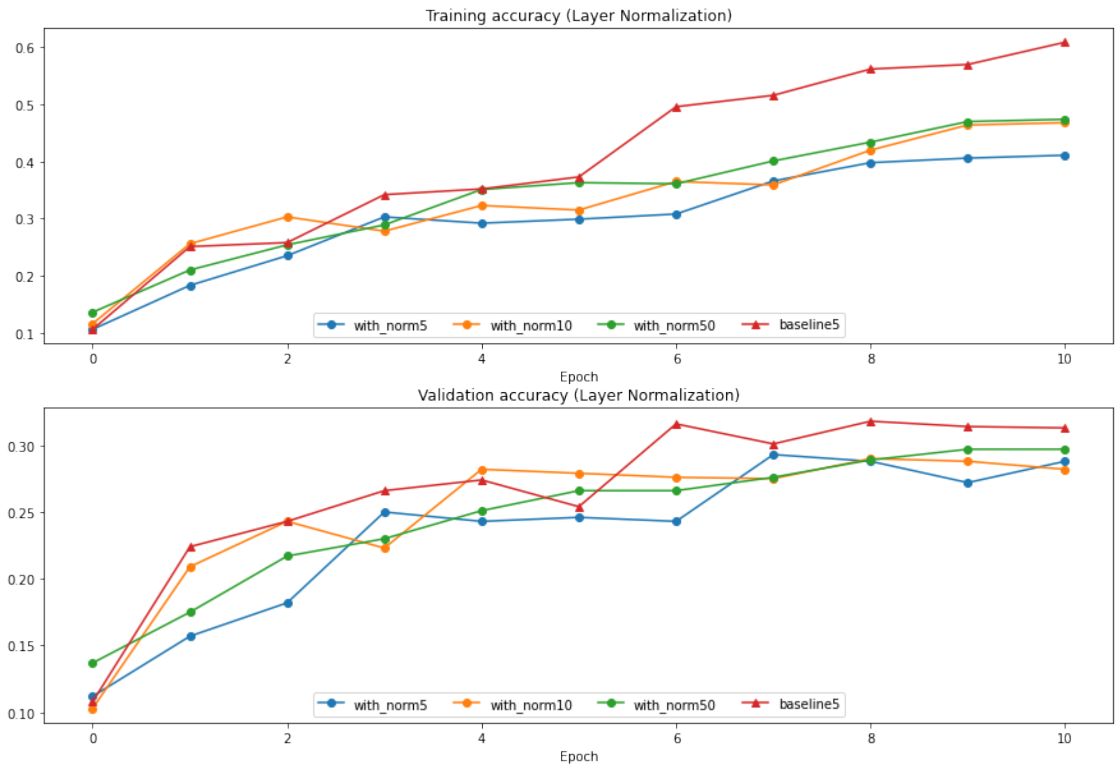
11 Layer Normalization and Batch Size

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

[53]:
```
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:

As data is vectorized per layer, having more layers won't really affect the performance of the normalization.

Smaller dimension corresponds to fewer values in one vector . This won't be able to square off the mean and variance of the pixel well, resulting in poor performance

Having high lambda can cause the weights to push further, which won't allow accurate weights on specific feature causing poor performance.

Dropout

November 5, 2022

[ ]:
```
# 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/assignment1/'
FOLDERNAME = 'cs231n/assignment2'
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
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignment2
```

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](https://arxiv.org/abs/1207.0580)

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

[ ]:
```
# 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.

[ ]:
```
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.

[ ]:
```
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:  1.892896954038074e-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:

[FILL THIS IN] Training using dropout, we know only p% of values are passed to the network and the rest are zeroed. And during test time , dropout scaling is off and hence the inputs will be different to the neurons. To make the input to the neurons same at test and train, we add a inverse dropout by p in dropout layer.

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.

[ ]:
```
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:  2.300479089768492
W1 relative error: 1.03e-07
W2 relative error: 2.21e-05
W3 relative error: 4.56e-07
b1 relative error: 1.00e+00
b2 relative error: 1.00e+00
b3 relative error: 1.00e+00

Running check with dropout =  0.75
Initial loss:  2.3024541059293124
W1 relative error: 5.98e-07
W2 relative error: 4.29e-06
W3 relative error: 4.91e-08
b1 relative error: 1.00e+00
b2 relative error: 1.00e+00
b3 relative error: 1.00e+00

Running check with dropout =  0.5
Initial loss:  2.3030067920900876
W1 relative error: 7.76e-07
W2 relative error: 9.41e-08
W3 relative error: 6.48e-07
b1 relative error: 1.00e+00
b2 relative error: 1.00e+00
b3 relative error: 1.00e+00
```

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.

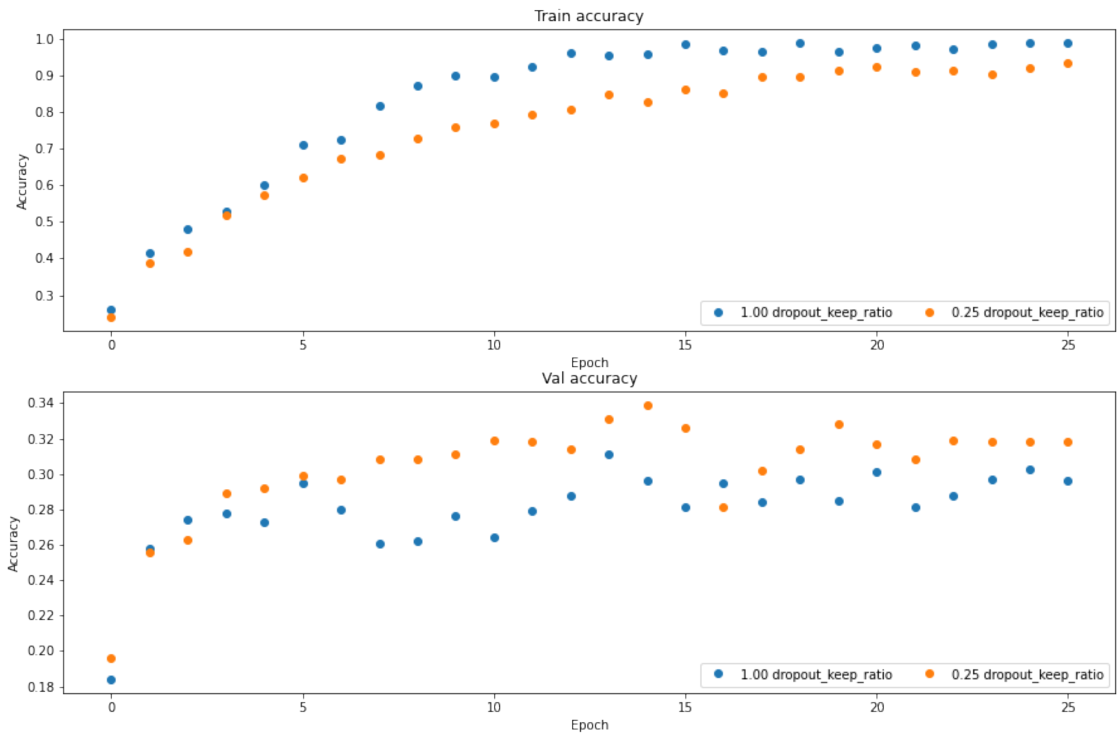
[ ]:
```python
# Train two identical nets, one with dropout and one without.
np.random.seed(231)
num_train = 500
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [1, 0.25]
for dropout_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: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.274000
(Epoch 3 / 25) train acc: 0.530000; val_acc: 0.278000
(Epoch 4 / 25) train acc: 0.602000; val_acc: 0.273000
(Epoch 5 / 25) train acc: 0.710000; val_acc: 0.295000
(Epoch 6 / 25) train acc: 0.724000; val_acc: 0.280000
(Epoch 7 / 25) train acc: 0.818000; val_acc: 0.261000
(Epoch 8 / 25) train acc: 0.872000; val_acc: 0.262000
(Epoch 9 / 25) train acc: 0.898000; val_acc: 0.276000
(Epoch 10 / 25) train acc: 0.896000; val_acc: 0.264000
(Epoch 11 / 25) train acc: 0.924000; val_acc: 0.279000
```

```
(Epoch 12 / 25) train acc: 0.962000; val_acc: 0.288000
(Epoch 13 / 25) train acc: 0.954000; val_acc: 0.311000
(Epoch 14 / 25) train acc: 0.958000; val_acc: 0.296000
(Epoch 15 / 25) train acc: 0.986000; val_acc: 0.281000
(Epoch 16 / 25) train acc: 0.968000; val_acc: 0.295000
(Epoch 17 / 25) train acc: 0.964000; val_acc: 0.284000
(Epoch 18 / 25) train acc: 0.988000; val_acc: 0.297000
(Epoch 19 / 25) train acc: 0.966000; val_acc: 0.285000
(Epoch 20 / 25) train acc: 0.976000; val_acc: 0.301000
(Iteration 101 / 125) loss: 0.271576
(Epoch 21 / 25) train acc: 0.980000; val_acc: 0.281000
(Epoch 22 / 25) train acc: 0.970000; val_acc: 0.288000
(Epoch 23 / 25) train acc: 0.984000; val_acc: 0.297000
(Epoch 24 / 25) train acc: 0.988000; val_acc: 0.303000
(Epoch 25 / 25) train acc: 0.988000; val_acc: 0.296000

0.25
(Iteration 1 / 125) loss: 4.897379
(Epoch 0 / 25) train acc: 0.240000; val_acc: 0.196000
(Epoch 1 / 25) train acc: 0.388000; val_acc: 0.256000
(Epoch 2 / 25) train acc: 0.420000; val_acc: 0.263000
(Epoch 3 / 25) train acc: 0.518000; val_acc: 0.289000
(Epoch 4 / 25) train acc: 0.572000; val_acc: 0.292000
(Epoch 5 / 25) train acc: 0.622000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.674000; val_acc: 0.297000
(Epoch 7 / 25) train acc: 0.682000; val_acc: 0.308000
(Epoch 8 / 25) train acc: 0.726000; val_acc: 0.308000
(Epoch 9 / 25) train acc: 0.760000; val_acc: 0.311000
(Epoch 10 / 25) train acc: 0.770000; val_acc: 0.319000
(Epoch 11 / 25) train acc: 0.794000; val_acc: 0.318000
(Epoch 12 / 25) train acc: 0.808000; val_acc: 0.314000
(Epoch 13 / 25) train acc: 0.848000; val_acc: 0.331000
(Epoch 14 / 25) train acc: 0.826000; val_acc: 0.339000
(Epoch 15 / 25) train acc: 0.862000; val_acc: 0.326000
(Epoch 16 / 25) train acc: 0.850000; val_acc: 0.281000
(Epoch 17 / 25) train acc: 0.894000; val_acc: 0.302000
(Epoch 18 / 25) train acc: 0.896000; val_acc: 0.314000
(Epoch 19 / 25) train acc: 0.912000; val_acc: 0.328000
(Epoch 20 / 25) train acc: 0.922000; val_acc: 0.317000
(Iteration 101 / 125) loss: 1.175938
(Epoch 21 / 25) train acc: 0.908000; val_acc: 0.308000
(Epoch 22 / 25) train acc: 0.914000; val_acc: 0.319000
(Epoch 23 / 25) train acc: 0.902000; val_acc: 0.318000
(Epoch 24 / 25) train acc: 0.920000; val_acc: 0.318000
(Epoch 25 / 25) train acc: 0.932000; val_acc: 0.318000
```

[ ]:

```
# 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:

[FILL THIS IN] Dropout reduces overfitting and during training, the model is perfroming better without dropout and reached lesser accuracy in val set than train set. Model with a dropout reaches higher accuracy in val and reduces gap in accuracy between train and val.

5.3 Inline Question 3:

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

5.4 Answer:

[FILL THIS IN] Higher the p for lesser depth networks and as depth of the networks incease, we can decrease the p value for dropout.

[ ]:

```

```

ConvolutionalNetworks

November 5, 2022

[5]:
```
# 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/assignment1/'
FOLDERNAME = 'cs231n/assignment2'
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
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignment2
```

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))
```

```
(2, 3, 6, 6)
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

2.1 Aside: Image Processing via Convolutions

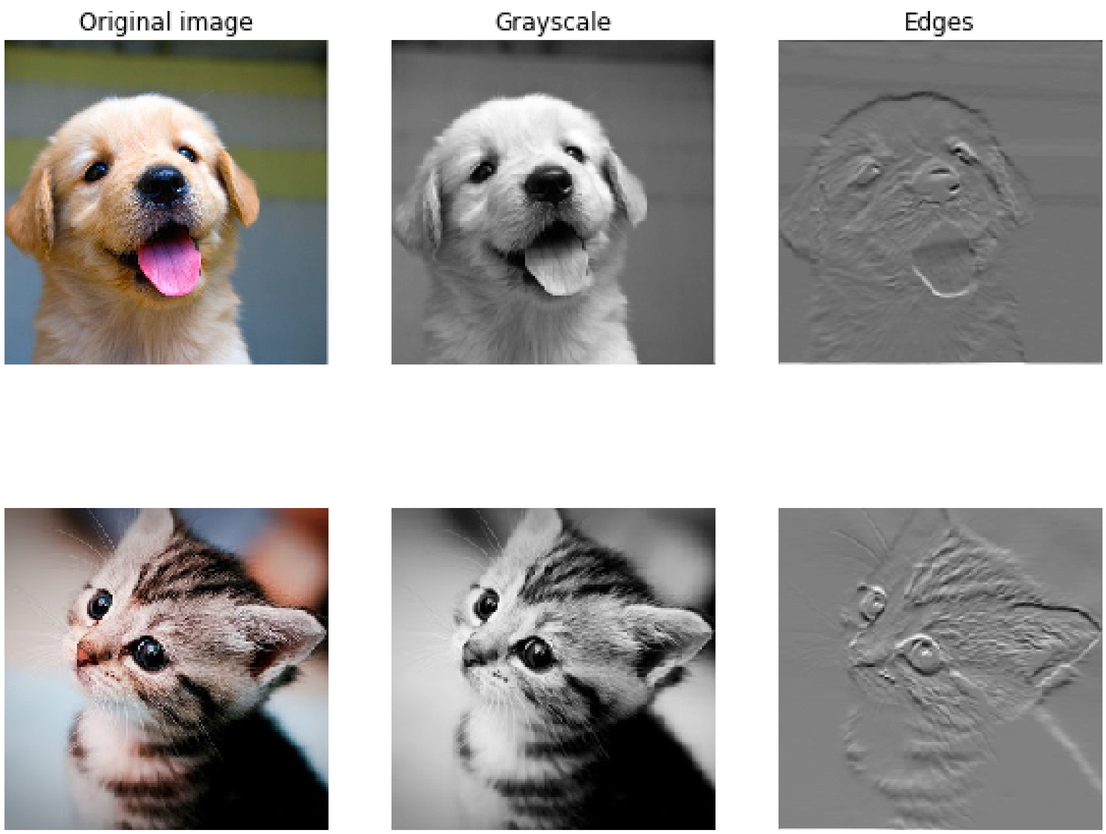
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection).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

```
(2, 3, 202, 202)
```



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.

[7]:
```
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:

[35]:
```
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:

[5]:
```
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.

[5]:
```
# 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/
```

```
/content/drive/My Drive/cs231n/assignment2/cs231n
running build_ext
/content/drive/My Drive/cs231n/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:

[6]:
```
# 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: 5.355578s
Fast: 0.010016s
Speedup: 534.704213x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 7.707152s
Fast: 0.013143s
Speedup: 586.415238x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  0.0
```

[7]:

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

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

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

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

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

```
Testing pool_forward_fast:
Naive: 0.487422s
fast: 0.006575s
speedup: 74.128794x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.604330s
fast: 0.019732s
speedup: 30.626178x
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.

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

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

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

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

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

```
Testing conv_relu_pool
dx error:  9.591132621921372e-09
dw error:  5.802391137330214e-09
db error:  1.0146343411762047e-09
```

[9]:
```
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.

[34]:
```
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):  2.3025850635890874
Initial loss (with regularization):  2.508600062137286
```

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.

[43]:
```
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: 3.422399e-04
b1 max relative error: 3.397321e-06
b2 max relative error: 1.000000e+00
b3 max relative error: 1.000000e+00
```

8.3 Overfit Small Data

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

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

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

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(
    model,
    small_data,
    num_epochs=15,
    batch_size=50,
```

```
    update_rule='adam',
    optim_config={'learning_rate': 1e-3,},
    verbose=True,
    print_every=1
)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.101625
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.269999
(Iteration 4 / 30) loss: 2.096722
(Epoch 2 / 15) train acc: 0.230000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.840178
(Iteration 6 / 30) loss: 1.934360
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.171000
(Iteration 7 / 30) loss: 1.827025
(Iteration 8 / 30) loss: 1.633108
(Epoch 4 / 15) train acc: 0.530000; val_acc: 0.181000
(Iteration 9 / 30) loss: 1.328687
(Iteration 10 / 30) loss: 1.759363
(Epoch 5 / 15) train acc: 0.620000; val_acc: 0.165000
(Iteration 11 / 30) loss: 1.031340
(Iteration 12 / 30) loss: 1.059685
(Epoch 6 / 15) train acc: 0.730000; val_acc: 0.227000
(Iteration 13 / 30) loss: 1.145554
(Iteration 14 / 30) loss: 0.851463
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.249000
(Iteration 15 / 30) loss: 0.584450
(Iteration 16 / 30) loss: 0.651959
(Epoch 8 / 15) train acc: 0.830000; val_acc: 0.243000
(Iteration 17 / 30) loss: 0.793184
(Iteration 18 / 30) loss: 0.472870
(Epoch 9 / 15) train acc: 0.840000; val_acc: 0.175000
(Iteration 19 / 30) loss: 0.442074
(Iteration 20 / 30) loss: 0.642406
(Epoch 10 / 15) train acc: 0.920000; val_acc: 0.195000
(Iteration 21 / 30) loss: 0.340764
(Iteration 22 / 30) loss: 0.267586
(Epoch 11 / 15) train acc: 0.810000; val_acc: 0.204000
(Iteration 23 / 30) loss: 0.480504
(Iteration 24 / 30) loss: 0.485876
(Epoch 12 / 15) train acc: 0.940000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111396
(Iteration 26 / 30) loss: 0.133215
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.209000
```

```
(Iteration 27 / 30) loss: 0.167909
(Iteration 28 / 30) loss: 0.228863
(Epoch 14 / 15) train acc: 0.950000; val_acc: 0.201000
(Iteration 29 / 30) loss: 0.160826
(Iteration 30 / 30) loss: 0.093452
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.222000
```

[19]:

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

```
Small data training accuracy: 0.79
```

[20]:

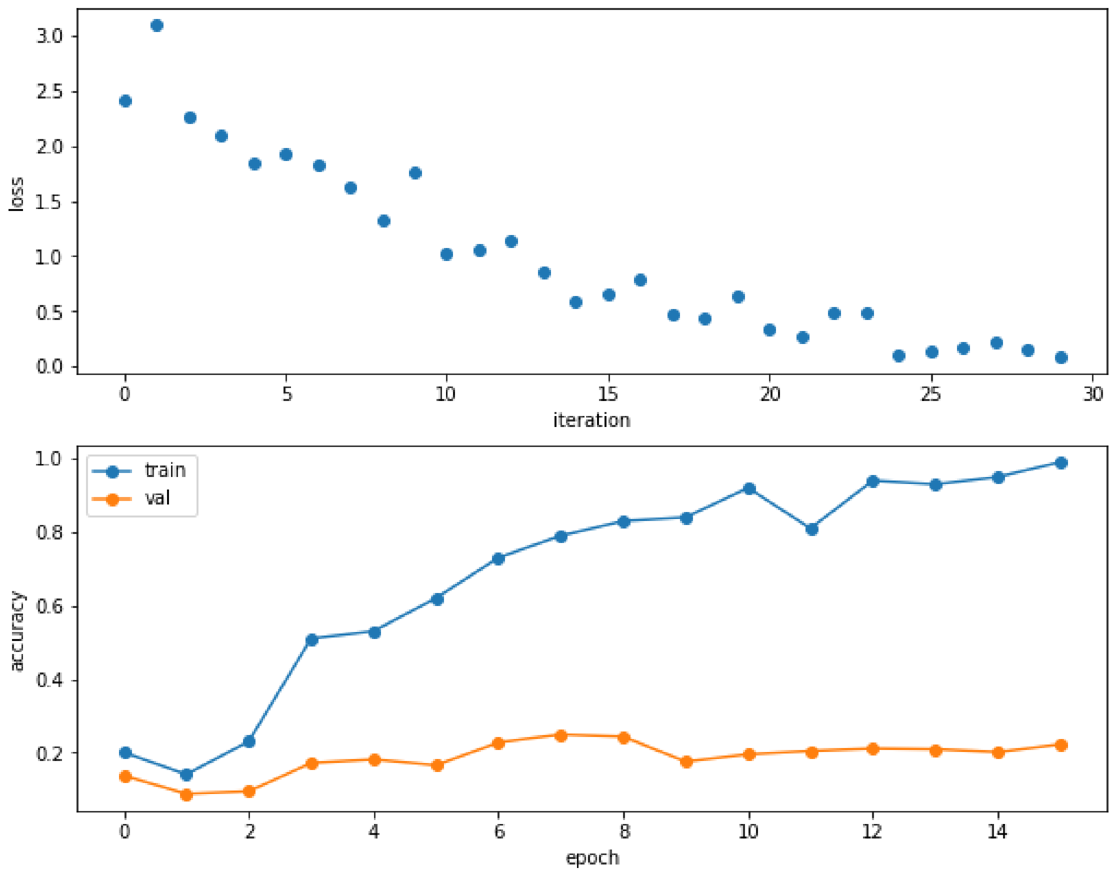
```
# Print final validation accuracy.
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

```
Small data validation accuracy: 0.249
```

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

[21]:

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

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



8.4 Train the Network

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

[44]:
```
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: 2.304640
(Epoch 0 / 1) train acc: 0.104000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.056490
(Iteration 41 / 980) loss: 2.038898
(Iteration 61 / 980) loss: 1.843205
(Iteration 81 / 980) loss: 2.028490
(Iteration 101 / 980) loss: 1.805907
(Iteration 121 / 980) loss: 1.910375
(Iteration 141 / 980) loss: 2.208175
(Iteration 161 / 980) loss: 1.757393
(Iteration 181 / 980) loss: 1.559073
(Iteration 201 / 980) loss: 1.832364
(Iteration 221 / 980) loss: 1.497918
(Iteration 241 / 980) loss: 1.298081
(Iteration 261 / 980) loss: 1.601285
(Iteration 281 / 980) loss: 1.688402
(Iteration 301 / 980) loss: 1.616132
(Iteration 321 / 980) loss: 1.931079
(Iteration 341 / 980) loss: 1.645593
(Iteration 361 / 980) loss: 1.829552
(Iteration 381 / 980) loss: 1.955372
(Iteration 401 / 980) loss: 1.539195
(Iteration 421 / 980) loss: 1.489417
(Iteration 441 / 980) loss: 1.715348
(Iteration 461 / 980) loss: 1.676086
(Iteration 481 / 980) loss: 1.681738
(Iteration 501 / 980) loss: 1.718939
(Iteration 521 / 980) loss: 1.432996
(Iteration 541 / 980) loss: 1.903807
(Iteration 561 / 980) loss: 1.688342
(Iteration 581 / 980) loss: 1.647393
(Iteration 601 / 980) loss: 1.305784
(Iteration 621 / 980) loss: 1.367091
(Iteration 641 / 980) loss: 1.577931
(Iteration 661 / 980) loss: 1.588563
(Iteration 681 / 980) loss: 1.612631
(Iteration 701 / 980) loss: 1.639757
(Iteration 721 / 980) loss: 1.591834
(Iteration 741 / 980) loss: 1.610143
(Iteration 761 / 980) loss: 1.618548
(Iteration 781 / 980) loss: 1.669676
(Iteration 801 / 980) loss: 1.422147
(Iteration 821 / 980) loss: 1.472864
(Iteration 841 / 980) loss: 1.580169
(Iteration 861 / 980) loss: 1.685842
(Iteration 881 / 980) loss: 1.191709
(Iteration 901 / 980) loss: 1.390020
(Iteration 921 / 980) loss: 1.712776
```

```
(Iteration 941 / 980) loss: 1.635240
(Iteration 961 / 980) loss: 1.596630
(Epoch 1 / 1) train acc: 0.483000; val_acc: 0.486000
```

[45]:
```
# Print final training accuracy.
print(
    "Full data training accuracy:",
    solver.check_accuracy(data['X_train'], data['y_train'])
)
```

```
Full data training accuracy: 0.4736530612244898
```

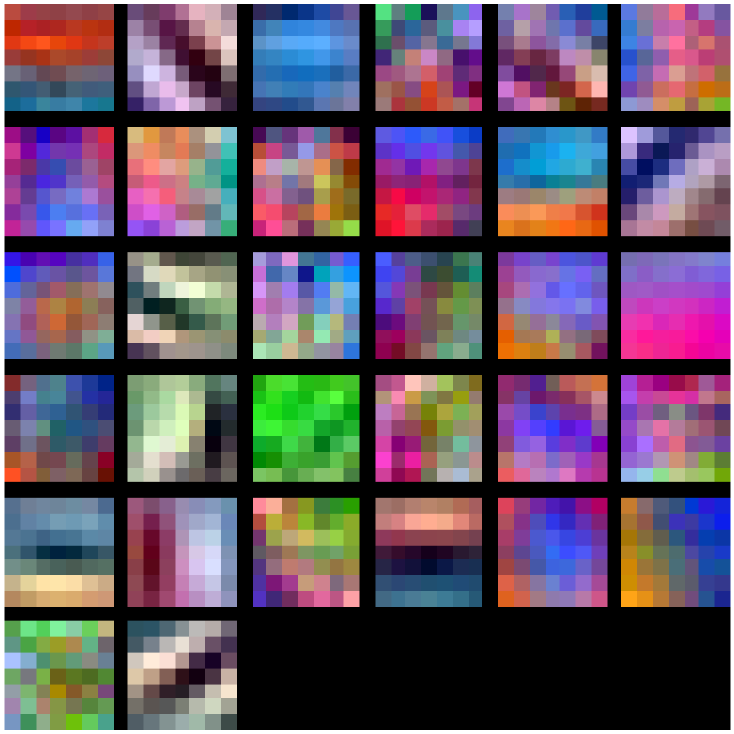
[46]:
```
# Print final validation accuracy.
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
```

```
Full data validation accuracy: 0.486
```

8.5 Visualize Filters

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

[47]:
```
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.](https://arxiv.org/abs/1502.03167)

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:

[48]:
```
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:  [ 0.0610569 -0.0451249 -0.015932 ]
  stds:  [1.02618378 0.96241754 1.00731258]
After spatial batch normalization (nontrivial gamma, beta):
  shape:  (2, 3, 4, 5)
  means:  [7.24357007 6.72291657 7.03351336]
  stds:  [4.26595256 3.90798026 4.28859798]
```

[49]:
```
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:  [-0.07486696  0.08207072  0.05214071  0.03632024]
  stds:  [0.96993818 1.03118426 1.02820835 1.0016052 ]
```

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:

[50]:
```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

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

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

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  2.721986061450212e-07
dgamma error:  5.909587857500314e-12
dbeta error:  3.2756370464786835e-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.](https://arxiv.org/pdf/1607.06450.pdf)

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

[4] [N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.](https://ieeexplore.ieee.org/abstract/document/1467360/)

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:

[22]:
```
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:

[35]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

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

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

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

# You should expect errors of magnitudes between 1e-12 and 1e-07.
print('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:  7.413109332145332e-08
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12
```

PyTorch

November 5, 2022

[ ]:
```
# 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/assignment1/'
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 (or TensorFlow, if you choose to work with that notebook).

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 or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).

- 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! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- 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.

[ ]:
```
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)
```

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.

[ ]:
```
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)
```

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).

[ ]:
```
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()
```

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.

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

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
```

```
    PyTorch will take care of the backward pass for us.

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

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

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

    w1, w2 = params

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


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

two_layer_fc_test()
```

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!

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

    pass

    # *****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).

[ ]:
```
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()
```

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

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

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

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

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.

```
[ ]: 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.

```
[ ]: 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.

```
[ ]: 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)
```

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.

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

     pass

     # *****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)
```

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:

[ ]:
```
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()
```

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.

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

        pass
```

```
        # *****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)*****

        pass

        # *****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()
```

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.

[ ]:
```
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.

```
[ ]: 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.

[ ]:
```
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)
```

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.

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

pass

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

train_part34(model, optimizer)
```

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.

[ ]:
```
# 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)
```

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.

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

pass

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

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

train_part34(model, optimizer)
```

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!

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

pass

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

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

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:

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.

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

TensorFlow

November 5, 2022

[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/assignment1/'
FOLDERNAME = 'cs231n/assignment2'
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
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignment2
```

1 Introduction to TensorFlow

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, TensorFlow (or PyTorch, if you choose to work with that notebook).

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 or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- 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! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- 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 TensorFlow?

TensorFlow is a system for executing computational graphs over Tensor objects, with native support for performing backpropogation for its Variables. In it, we work with Tensors which are n-dimensional arrays analogous to the numpy ndarray.

1.3 How do I learn TensorFlow?

TensorFlow has many excellent tutorials available, including those from Google themselves.

Otherwise, this notebook will walk you through much of what you need to do to train models in TensorFlow. See the end of the notebook for some links to helpful tutorials if you want to learn more or need further clarification on topics that aren't fully explained here.

Note: This notebook is meant to teach you Tensorflow 2.x. Most examples on the web today are still in 1.x, so be careful not to confuse the two when looking up documentation.

2 Table of Contents

This notebook has 5 parts. We will walk through TensorFlow at **three different levels of abstraction**, which should help you better understand it and prepare you for working on your project.

1. Part I, Preparation: load the CIFAR-10 dataset.
2. Part II, Barebone TensorFlow: **Abstraction Level 1**, we will work directly with low-level TensorFlow graphs.
3. Part III, Keras Model API: **Abstraction Level 2**, we will use `tf.keras.Model` to define arbitrary neural network architecture.
4. Part IV, Keras Sequential + Functional API: **Abstraction Level 3**, we will use `tf.keras.Sequential` to define a linear feed-forward network very conveniently, and then

explore the functional libraries for building unique and uncommon models that require more flexibility.

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.

We will discuss Keras in more detail later in the notebook.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`tf.keras.Model`	High	Medium
`tf.keras.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.

[6]:
```
import os
import tensorflow as tf
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt

%matplotlib inline

USE_GPU = True

if USE_GPU:
    device = '/device:GPU:0'
else:
    device = '/cpu:0'

# Constant to control how often we print when training models.
print_every = 100
print('Using device: ', device)
```

```
Using device:  /device:GPU:0
```

4 Part I: Preparation

First, we load the CIFAR-10 dataset. This might take a few minutes to download the first time you run it, but after that the files should be cached on disk and loading should be faster.

In previous parts of the assignment we used CS231N-specific code to download and read the CIFAR-10 dataset; however the `tf.keras.datasets` package in TensorFlow provides prebuilt utility functions for loading many common datasets.

For the purposes of this assignment we will still write our own code to preprocess the data and iterate through it in minibatches. The `tf.data` package in TensorFlow provides tools for automating this process, but working with this package adds extra complication and is beyond the scope of this notebook. However using `tf.data` can be much more efficient than the simple approach used in this notebook, so you should consider using it for your project.

[7]:
```
def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
    """
    Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 dataset and use appropriate data types and shapes
    cifar10 = tf.keras.datasets.cifar10.load_data()
    (X_train, y_train), (X_test, y_test) = cifar10
    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.int32).flatten()
    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.int32).flatten()

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean pixel and divide by std
    mean_pixel = X_train.mean(axis=(0, 1, 2), keepdims=True)
    std_pixel = X_train.std(axis=(0, 1, 2), keepdims=True)
    X_train = (X_train - mean_pixel) / std_pixel
    X_val = (X_val - mean_pixel) / std_pixel
    X_test = (X_test - mean_pixel) / std_pixel

    return X_train, y_train, X_val, y_val, X_test, y_test
```

```
# If there are errors with SSL downloading involving self-signed certificates,
# it may be that your Python version was recently installed on the current␣
↪machine.
# See: https://github.com/tensorflow/tensorflow/issues/10779
# To fix, run the command: /Applications/Python\ 3.7/Install\ Certificates.
↪command
#   ...replacing paths as necessary.

# Invoke the above function to get our data.
NHW = (0, 1, 2)
X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape, y_train.dtype)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 14s 0us/step
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[8]:
```
class Dataset(object):
    def __init__(self, X, y, batch_size, shuffle=False):
        """
        Construct a Dataset object to iterate over data X and labels y

        Inputs:
        - X: Numpy array of data, of any shape
        - y: Numpy array of labels, of any shape but with y.shape[0] == X.
↪shape[0]
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
        """
        assert X.shape[0] == y.shape[0], 'Got different numbers of data and␣
↪labels'
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
```

```
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))


train_dset = Dataset(X_train, y_train, batch_size=64, shuffle=True)
val_dset = Dataset(X_val, y_val, batch_size=64, shuffle=False)
test_dset = Dataset(X_test, y_test, batch_size=64)
```

[9]:
```
# We can iterate through a dataset like this:
for t, (x, y) in enumerate(train_dset):
    print(t, x.shape, y.shape)
    if t > 5: break
```

```
0 (64, 32, 32, 3) (64,)
1 (64, 32, 32, 3) (64,)
2 (64, 32, 32, 3) (64,)
3 (64, 32, 32, 3) (64,)
4 (64, 32, 32, 3) (64,)
5 (64, 32, 32, 3) (64,)
6 (64, 32, 32, 3) (64,)
```

5 Part II: Barebones TensorFlow

TensorFlow ships with various high-level APIs which make it very convenient to define and train neural networks; we will cover some of these constructs in Part III and Part IV of this notebook. In this section we will start by building a model with basic TensorFlow constructs to help you better understand what's going on under the hood of the higher-level APIs.

"Barebones Tensorflow" is important to understanding the building blocks of TensorFlow, but much of it involves concepts from TensorFlow 1.x. We will be working with legacy modules such as `tf.Variable`.

Therefore, please read and understand the differences between legacy (1.x) TF and the new (2.0) TF.

5.0.1 Historical background on TensorFlow 1.x

TensorFlow 1.x is primarily a framework for working with **static computational graphs**. Nodes in the computational graph are Tensors which will hold n-dimensional arrays when the graph is run; edges in the graph represent functions that will operate on Tensors when the graph is run to actually perform useful computation.

Before Tensorflow 2.0, we had to configure the graph into two phases. There are plenty of tutorials online that explain this two-step process. The process generally looks like the following for TF 1.x: 1. **Build a computational graph that describes the computation that you want**

to perform. This stage doesn't actually perform any computation; it just builds up a symbolic representation of your computation. This stage will typically define one or more `placeholder` objects that represent inputs to the computational graph. 2. **Run the computational graph many times.** Each time the graph is run (e.g. for one gradient descent step) you will specify which parts of the graph you want to compute, and pass a `feed_dict` dictionary that will give concrete values to any `placeholder`s in the graph.

5.0.2 The new paradigm in Tensorflow 2.0

Now, with Tensorflow 2.0, we can simply adopt a functional form that is more Pythonic and similar in spirit to PyTorch and direct Numpy operation. Instead of the 2-step paradigm with computation graphs, making it (among other things) easier to debug TF code. You can read more details at https://www.tensorflow.org/guide/eager.

The main difference between the TF 1.x and 2.0 approach is that the 2.0 approach doesn't make use of `tf.Session`, `tf.run`, `placeholder`, `feed_dict`. To get more details of what's different between the two version and how to convert between the two, check out the official migration guide: https://www.tensorflow.org/alpha/guide/migration_guide

Later, in the rest of this notebook we'll focus on this new, simpler approach.

5.0.3 TensorFlow warmup: Flatten Function

We can see this in action by defining a simple `flatten` function that will reshape image data for use in a fully-connected network.

In TensorFlow, data for convolutional feature maps is typically stored in a Tensor of shape N x H x W x C where:

- N is the number of datapoints (minibatch size)
- H is the height of the feature map
- W is the width of the feature map
- C is the number of channels in the feature map

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 `H x W x C` values per representation into a single long vector.

Notice the `tf.reshape` call has the target shape as `(N, -1)`, meaning it will reshape/keep the first dimension to be N, and then infer as necessary what the second dimension is in the output, so we can collapse the remaining dimensions from the input properly.

NOTE: TensorFlow and PyTorch differ on the default Tensor layout; TensorFlow uses N x H x W x C but PyTorch uses N x C x H x W.

[10]:
```
def flatten(x):
    """
```

```
    Input:
    - TensorFlow Tensor of shape (N, D1, ..., DM)

    Output:
    - TensorFlow Tensor of shape (N, D1 * ... * DM)
    """
    N = tf.shape(x)[0]
    return tf.reshape(x, (N, -1))
```

[11]:
```
def test_flatten():
    # Construct concrete values of the input data x using numpy
    x_np = np.arange(24).reshape((2, 3, 4))
    print('x_np:\n', x_np, '\n')
    # Compute a concrete output value.
    x_flat_np = flatten(x_np)
    print('x_flat_np:\n', x_flat_np, '\n')

test_flatten()
```

```
x_np:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

x_flat_np:
 tf.Tensor(
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]], shape=(2, 12), dtype=int64)
```

5.0.4 Barebones TensorFlow: Define a Two-Layer Network

We will now implement our first neural network with TensorFlow: a fully-connected ReLU network with two hidden layers and no biases on the CIFAR10 dataset. For now we will use only low-level TensorFlow operators to define the network; later we will see how to use the higher-level abstractions provided by `tf.keras` to simplify the process.

We will define the forward pass of the network in the function `two_layer_fc`; this will accept TensorFlow Tensors for the inputs and weights of the network, and return a TensorFlow Tensor for the scores.

After defining the network architecture in the `two_layer_fc` function, we will test the implementation by checking the shape of the output.

It's important that you read and understand this implementation.

[12]:
```python
def two_layer_fc(x, params):
    """
    A fully-connected neural network; the architecture is:
    fully-connected layer -> ReLU -> fully connected layer.
    Note that we only need to define the forward pass here; TensorFlow will take
    care of computing the gradients 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 TensorFlow Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of TensorFlow Tensors giving weights for the
      network, where w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A TensorFlow Tensor of shape (N, C) giving classification scores
      for the input data x.
    """
    w1, w2 = params                   # Unpack the parameters
    x = flatten(x)                    # Flatten the input; now x has shape (N,␣
↪D)
    h = tf.nn.relu(tf.matmul(x, w1))  # Hidden layer: h has shape (N, H)
    scores = tf.matmul(h, w2)         # Compute scores of shape (N, C)
    return scores
```

[13]:
```python
def two_layer_fc_test():
    hidden_layer_size = 42

    # Scoping our TF operations under a tf.device context manager
    # lets us tell TensorFlow where we want these Tensors to be
    # multiplied and/or operated on, e.g. on a CPU or a GPU.
    with tf.device(device):
        x = tf.zeros((64, 32, 32, 3))
        w1 = tf.zeros((32 * 32 * 3, hidden_layer_size))
        w2 = tf.zeros((hidden_layer_size, 10))

        # Call our two_layer_fc function for the forward pass of the network.
        scores = two_layer_fc(x, [w1, w2])

    print(scores.shape)
```

```
two_layer_fc_test()
```

```
(64, 10)
```

5.0.5 Barebones TensorFlow: 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. 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.

HINT: For convolutions: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/nn/conv2d; be careful with padding!

HINT: For biases: https://www.tensorflow.org/performance/xla/broadcasting

[68]:
```
def three_layer_convnet(x, params):
    """
    A three-layer convolutional network with the architecture described above.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of images
    - params: A list of TensorFlow Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
        weights for the first convolutional layer.
      - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
        first convolutional layer.
      - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
        giving weights for the second convolutional layer
      - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
        second convolutional layer.
      - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
        Can you figure out what the shape should be?
      - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
        Can you figure out what the shape should be?
    """
    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)*****
    #We know that forward pass of the conv  layer consists of Conv -> ReLu ->␣
↪Pool
    #Implementing this using tensorflow, we get 3 initializations
    #Conv2D, ReLU and bias term
    #First we initialize zero padding for 2
    x = tf.pad(x, tf.constant(((0,0), (2,2), (2,2), (0,0))))
    #performing y=wx+b
    conv1=tf.nn.conv2d(x, conv_w1, strides=[1,1,1,1], padding='VALID',␣
↪data_format='NHWC', dilations=None, name=None)+conv_b1
    #Perfroming ReLU once
    relu1=tf.nn.relu(conv1)

    #Repeating the same for the next layer
    conv1 = tf.pad(conv1, tf.constant(((0,0), (1,1), (1,1), (0,0))))
    conv=tf.nn.conv2d(conv1, conv_w2, strides=[1,1,1,1], padding='VALID',␣
↪data_format='NHWC')+conv_b2
    #Perfroming ReLU once
    relu=tf.nn.relu(conv)

    #Calculating the final score, before that we flatten the data in H W C␣
↪format to H*W*C format
    f=flatten(relu)
    scores=tf.matmul(f,fc_w)+fc_b


    pass

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

After defing the forward pass of the three-layer ConvNet above, run the following cell to test your implementation. Like the two-layer network, we run the graph on a batch of zeros just to make sure the function doesn't crash, and produces outputs of the correct shape.

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

[69]:
```
def three_layer_convnet_test():

    with tf.device(device):
        x = tf.zeros((64, 32, 32, 3))
        conv_w1 = tf.zeros((5, 5, 3, 6))
        conv_b1 = tf.zeros((6,))
```

```
        conv_w2 = tf.zeros((3, 3, 6, 9))
        conv_b2 = tf.zeros((9,))
        fc_w = tf.zeros((32 * 32 * 9, 10))
        fc_b = tf.zeros((10,))
        params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
        scores = three_layer_convnet(x, params)

    # Inputs to convolutional layers are 4-dimensional arrays with shape
    # [batch_size, height, width, channels]
    print('scores_np has shape: ', scores.shape)

three_layer_convnet_test()
```

```
scores_np has shape:  (64, 10)
```

5.0.6 Barebones TensorFlow: Training Step

We now define the `training_step` function performs a single training step. This will take three basic steps:

1. Compute the loss
2. Compute the gradient of the loss with respect to all network weights
3. Make a weight update step using (stochastic) gradient descent.

We need to use a few new TensorFlow functions to do all of this: - For computing the cross-entropy loss we'll use `tf.nn.sparse_softmax_cross_entropy_with_logits`: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits

- For averaging the loss across a minibatch of data we'll use `tf.reduce_mean`: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/reduce_mean
- For computing gradients of the loss with respect to the weights we'll use `tf.GradientTape` (useful for Eager execution): https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/GradientTape
- We'll mutate the weight values stored in a TensorFlow Tensor using `tf.assign_sub` ("sub" is for subtraction): https://www.tensorflow.org/api_docs/python/tf/assign_sub

[16]:
```
def training_step(model_fn, x, y, params, learning_rate):
    with tf.GradientTape() as tape:
        scores = model_fn(x, params) # Forward pass of the model
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,␣
 ↪logits=scores)
        total_loss = tf.reduce_mean(loss)
        grad_params = tape.gradient(total_loss, params)

        # Make a vanilla gradient descent step on all of the model parameters
        # Manually update the weights using assign_sub()
        for w, grad_w in zip(params, grad_params):
```

```
            w.assign_sub(learning_rate * grad_w)

        return total_loss
```

[17]:
```
def train_part2(model_fn, init_fn, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model
      using TensorFlow; it should have the following signature:
      scores = model_fn(x, params) where x is a TensorFlow Tensor giving a
      minibatch of image data, params is a list of TensorFlow Tensors holding
      the model weights, and scores is a TensorFlow Tensor of shape (N, C)
      giving scores for all elements of x.
    - init_fn: A Python function that initializes the parameters of the model.
      It should have the signature params = init_fn() where params is a list
      of TensorFlow Tensors holding the (randomly initialized) weights of the
      model.
    - learning_rate: Python float giving the learning rate to use for SGD.
    """


    params = init_fn()  # Initialize the model parameters

    for t, (x_np, y_np) in enumerate(train_dset):
        # Run the graph on a batch of training data.
        loss = training_step(model_fn, x_np, y_np, params, learning_rate)

        # Periodically print the loss and check accuracy on the val set.
        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss))
            check_accuracy(val_dset, x_np, model_fn, params)
```

[18]:
```
def check_accuracy(dset, x, model_fn, params):
    """
    Check accuracy on a classification model, e.g. for validation.

    Inputs:
    - dset: A Dataset object against which to check accuracy
    - x: A TensorFlow placeholder Tensor where input images should be fed
    - model_fn: the Model we will be calling to make predictions on x
    - params: parameters for the model_fn to work with

    Returns: Nothing, but prints the accuracy of the model
    """
    num_correct, num_samples = 0, 0
```

```
    for x_batch, y_batch in dset:
        scores_np = model_fn(x_batch, params).numpy()
        y_pred = scores_np.argmax(axis=1)
        num_samples += x_batch.shape[0]
        num_correct += (y_pred == y_batch).sum()
    acc = float(num_correct) / num_samples
    print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *␣
↪acc))
```

5.0.7 Barebones TensorFlow: Initialization

We'll use the following utility method to initialize the weight matrices for our models using Kaiming's normalization method.

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

[19]:
```
def create_matrix_with_kaiming_normal(shape):
    if len(shape) == 2:
        fan_in, fan_out = shape[0], shape[1]
    elif len(shape) == 4:
        fan_in, fan_out = np.prod(shape[:3]), shape[3]
    return tf.keras.backend.random_normal(shape) * np.sqrt(2.0 / fan_in)
```

5.0.8 Barebones TensorFlow: Train a Two-Layer Network

We are finally ready to use all of the pieces defined above to train a two-layer fully-connected network on CIFAR-10.

We just need to define a function to initialize the weights of the model, and call `train_part2`.

Defining the weights of the network introduces another important piece of TensorFlow API: `tf.Variable`. A TensorFlow Variable is a Tensor whose value is stored in the graph and persists across runs of the computational graph; however unlike constants defined with `tf.zeros` or `tf.random_normal`, the values of a Variable can be mutated as the graph runs; these mutations will persist across graph runs. Learnable parameters of the network are usually stored in Variables.

You don't need to tune any hyperparameters, but you should achieve validation accuracies above 40% after one epoch of training.

[20]:
```
def two_layer_fc_init():
    """
    Initialize the weights of a two-layer network, for use with the
    two_layer_network function defined above.
    You can use the `create_matrix_with_kaiming_normal` helper!

    Inputs: None
```

```
    Returns: A list of:
    - w1: TensorFlow tf.Variable giving the weights for the first layer
    - w2: TensorFlow tf.Variable giving the weights for the second layer
    """
    hidden_layer_size = 4000
    w1 = tf.Variable(create_matrix_with_kaiming_normal((3 * 32 * 32, 4000)))
    w2 = tf.Variable(create_matrix_with_kaiming_normal((4000, 10)))
    return [w1, w2]

learning_rate = 1e-2
train_part2(two_layer_fc, two_layer_fc_init, learning_rate)
```

```
Iteration 0, loss = 2.9501
Got 146 / 1000 correct (14.60%)
Iteration 100, loss = 1.9061
Got 375 / 1000 correct (37.50%)
Iteration 200, loss = 1.4873
Got 392 / 1000 correct (39.20%)
Iteration 300, loss = 1.8757
Got 373 / 1000 correct (37.30%)
Iteration 400, loss = 1.7538
Got 418 / 1000 correct (41.80%)
Iteration 500, loss = 1.7386
Got 440 / 1000 correct (44.00%)
Iteration 600, loss = 1.8496
Got 429 / 1000 correct (42.90%)
Iteration 700, loss = 2.0136
Got 440 / 1000 correct (44.00%)
```

5.0.9 Barebones TensorFlow: Train a three-layer ConvNet

We will now use TensorFlow to train a three-layer ConvNet on CIFAR-10.

You need to implement the `three_layer_convnet_init` function. Recall that the architecture of the network is:

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

You don't need to do any hyperparameter tuning, but you should see validation accuracies above 43% after one epoch of training.

[70]:
```
def three_layer_convnet_init():
    """
```

```
    Initialize the weights of a Three-Layer ConvNet, for use with the
    three_layer_convnet function defined above.
    You can use the `create_matrix_with_kaiming_normal` helper!

    Inputs: None

    Returns a list containing:
    - conv_w1: TensorFlow tf.Variable giving weights for the first conv layer
    - conv_b1: TensorFlow tf.Variable giving biases for the first conv layer
    - conv_w2: TensorFlow tf.Variable giving weights for the second conv layer
    - conv_b2: TensorFlow tf.Variable giving biases for the second conv layer
    - fc_w: TensorFlow tf.Variable giving weights for the fully-connected layer
    - fc_b: TensorFlow tf.Variable giving biases for the fully-connected layer
    """
    params = None
    ############################################################################
    # TODO: Initialize the parameters of the three-layer network.             #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #Here we need to initialize weights and bias for a conv net with dimensions␣
↪mentioned above
    #For the first conv layer, it is mentioned as 32 filters of size 5x5x3
    conv_w1 = tf.Variable(create_matrix_with_kaiming_normal([5,5,3,32]))
    #we create bias terms for these 32 filters using tf.Variable
    conv_b1=tf.Variable(np.zeros([32]),dtype=tf.float32)
    #Peforming similar operation for the second filter we get
    conv_w2 = tf.Variable(create_matrix_with_kaiming_normal([3,3,3,16]))
    conv_b2=tf.Variable(np.zeros([16]),dtype=tf.float32)
    #Now, we need to initialize weights and bias for FC Layer. the flattened␣
↪layer will be multiple of 32x32x16 and for 10 classes
    fc_w = tf.Variable(create_matrix_with_kaiming_normal([32*32*16,10]))
    fc_b = tf.Variable(np.zeros([10]), dtype=tf.float32)
    params=(conv_w1,conv_b1,conv_w2,conv_b2,fc_b,fc_w)

    pass

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

learning_rate = 3e-3
train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate)
```

```
Iteration 0, loss = 3.0078
Got 144 / 1000 correct (14.40%)
```

```
Iteration 100, loss = 1.7259
Got 380 / 1000 correct (38.00%)
Iteration 200, loss = 1.4726
Got 411 / 1000 correct (41.10%)
Iteration 300, loss = 1.7055
Got 405 / 1000 correct (40.50%)
Iteration 400, loss = 1.6590
Got 446 / 1000 correct (44.60%)
Iteration 500, loss = 1.6420
Got 464 / 1000 correct (46.40%)
Iteration 600, loss = 1.5379
Got 474 / 1000 correct (47.40%)
Iteration 700, loss = 1.6041
Got 484 / 1000 correct (48.40%)
```

6 Part III: Keras Model Subclassing API

Implementing a neural network using the low-level TensorFlow API is a good way to understand how TensorFlow works, but it's a little inconvenient - we had to manually keep track of all Tensors holding learnable parameters. This was fine for a small network, but could quickly become unweildy for a large complex model.

Fortunately TensorFlow 2.0 provides higher-level APIs such as `tf.keras` which make it easy to build models out of modular, object-oriented layers. Further, TensorFlow 2.0 uses eager execution that evaluates operations immediately, without explicitly constructing any computational graphs. This makes it easy to write and debug models, and reduces the boilerplate code.

In this part of the notebook we will define neural network models using the `tf.keras.Model` API. To implement your own model, you need to do the following:

1. Define a new class which subclasses `tf.keras.Model`. Give your class an intuitive name that describes it, like `TwoLayerFC` or `ThreeLayerConvNet`.
2. In the initializer `__init__()` for your new class, define all the layers you need as class attributes. The `tf.keras.layers` package provides many common neural-network layers, like `tf.keras.layers.Dense` for fully-connected layers and `tf.keras.layers.Conv2D` for convolutional layers. Under the hood, these layers will construct `Variable` Tensors for any learnable parameters. **Warning**: Don't forget to call `super(YourModelName, self).__init__()` as the first line in your initializer!
3. Implement the `call()` method for your class; this implements the forward pass of your model, and defines the *connectivity* of your network. Layers defined in `__init__()` implement `__call__()` so they can be used as function objects that transform input Tensors into output Tensors. Don't define any new layers in `call()`; any layers you want to use in the forward pass should be defined in `__init__()`.

After you define your `tf.keras.Model` subclass, you can instantiate it and use it like the model functions from Part II.

6.0.1 Keras Model Subclassing API: Two-Layer Network

Here is a concrete example of using the `tf.keras.Model` API to define a two-layer network. There are a few new bits of API to be aware of here:

We use an `Initializer` object to set up the initial values of the learnable parameters of the layers; in particular `tf.initializers.VarianceScaling` gives behavior similar to the Kaiming initialization method we used in Part II. You can read more about it here: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/initializers/VarianceScaling

We construct `tf.keras.layers.Dense` objects to represent the two fully-connected layers of the model. In addition to multiplying their input by a weight matrix and adding a bias vector, these layer can also apply a nonlinearity for you. For the first layer we specify a ReLU activation function by passing `activation='relu'` to the constructor; the second layer uses softmax activation function. Finally, we use `tf.keras.layers.Flatten` to flatten the output from the previous fully-connected layer.

[71]:
```
class TwoLayerFC(tf.keras.Model):
    def __init__(self, hidden_size, num_classes):
        super(TwoLayerFC, self).__init__()
        initializer = tf.initializers.VarianceScaling(scale=2.0)
        self.fc1 = tf.keras.layers.Dense(hidden_size, activation='relu',
                                   kernel_initializer=initializer)
        self.fc2 = tf.keras.layers.Dense(num_classes, activation='softmax',
                                   kernel_initializer=initializer)
        self.flatten = tf.keras.layers.Flatten()

    def call(self, x, training=False):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x


def test_TwoLayerFC():
    """ A small unit test to exercise the TwoLayerFC model above. """
    input_size, hidden_size, num_classes = 50, 42, 10
    x = tf.zeros((64, input_size))
    model = TwoLayerFC(hidden_size, num_classes)
    with tf.device(device):
        scores = model(x)
        print(scores.shape)

test_TwoLayerFC()
```

```
(64, 10)
```

6.0.2 Keras Model Subclassing API: Three-Layer ConvNet

Now it's your turn to implement a three-layer ConvNet using the `tf.keras.Model` API. Your model should have the same architecture used in Part II:

1. Convolutional layer with 5 x 5 kernels, with zero-padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 3 x 3 kernels, with zero-padding of 1
4. ReLU nonlinearity
5. Fully-connected layer to give class scores
6. Softmax nonlinearity

You should initialize the weights of your network using the same initialization method as was used in the two-layer network above.

Hint: Refer to the documentation for `tf.keras.layers.Conv2D` and `tf.keras.layers.Dense`:

https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Conv2D

https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Dense

[75]:
```
class ThreeLayerConvNet(tf.keras.Model):
    def __init__(self, channel_1, channel_2, num_classes):
        super(ThreeLayerConvNet, self).__init__()
        ########################################################################
        # TODO: Implement the __init__ method for a three-layer ConvNet. You   #
        # should instantiate layer objects to be used in the forward pass.     #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        #Define filters and padding for all Conv operations first
        self.padding1=(2,2)
        self.padding2=(1,1)
        self.filter1=(5,5)
        self.filter2=(3,3)
        #Intializer for FC Net (As done in 2 FC net)
        initializer = tf.initializers.VarianceScaling(scale=2.0)

        #Initialize Zero Padding using https://www.tensorflow.org/api_docs/
↪python/tf/keras/layers/ZeroPadding2D4
        self.padding1=tf.keras.layers.ZeroPadding2D(padding=self.padding1)
        self.padding2=tf.keras.layers.ZeroPadding2D(padding=self.padding2)
        #Performing convilution for for the first kayer using valid padding as␣
↪mentioned
        self.conv1=tf.keras.layers.Conv2D(kernel_size = self.filter1, filters =␣
↪channel_1,
                                          padding = 'valid',␣
↪activation='relu')
        #Performing convilution for for the second layer using valid padding as␣
↪mentioned
```

```
        self.conv2=tf.keras.layers.Conv2D(kernel_size = self.filter2, filters =␣
↪channel_2,
                                              padding = 'valid',␣
↪activation='relu')
        #flattening the conv output
        self.flatten = tf.keras.layers.Flatten()

        self.fc = tf.keras.layers.Dense(num_classes, activation='softmax',
                                   kernel_initializer=initializer)
        pass

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

    def call(self, x, training=False):
        scores = None
        ########################################################################
        # TODO: Implement the forward pass for a three-layer ConvNet. You      #
        # should use the layer objects defined in the __init__ method.         #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
       #Calling back the class objects using x for first layer
        x = self.padding1(x)
        x = self.conv1(x)
       #Calling back the class objects using x for second layer as its forward␣
↪continous pass
        x = self.padding2(x)
        x = self.conv2(x)
      #Calling back the class objects using x for flattening layer
        x = self.flatten(x)
        #The final layer score is computed after the fc net which is then␣
↪stored as scores
        scores = self.fc(x)

        pass

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

Once you complete the implementation of the ThreeLayerConvNet above you can run the following to ensure that your implementation does not crash and produces outputs of the expected shape.

[76]:
```python
def test_ThreeLayerConvNet():
    channel_1, channel_2, num_classes = 12, 8, 10
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    with tf.device(device):
        x = tf.zeros((64, 3, 32, 32))
        scores = model(x)
        print(scores.shape)

test_ThreeLayerConvNet()
```

```
(64, 10)
```

6.0.3 Keras Model Subclassing API: Eager Training

While keras models have a builtin training loop (using the `model.fit`), sometimes you need more customization. Here's an example, of a training loop implemented with eager execution.

In particular, notice `tf.GradientTape`. Automatic differentiation is used in the backend for implementing backpropagation in frameworks like TensorFlow. During eager execution, `tf.GradientTape` is used to trace operations for computing gradients later. A particular `tf.GradientTape` can only compute one gradient; subsequent calls to tape will throw a runtime error.

TensorFlow 2.0 ships with easy-to-use built-in metrics under `tf.keras.metrics` module. Each metric is an object, and we can use `update_state()` to add observations and `reset_state()` to clear all observations. We can get the current result of a metric by calling `result()` on the metric object.

[77]:
```python
def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1,␣
 ↪is_training=False):
    """
    Simple training loop for use with models defined using tf.keras. It trains
    a model for one epoch on the CIFAR-10 training set and periodically checks
    accuracy on the CIFAR-10 validation set.

    Inputs:
    - model_init_fn: A function that takes no parameters; when called it
      constructs the model we want to train: model = model_init_fn()
    - optimizer_init_fn: A function which takes no parameters; when called it
      constructs the Optimizer object we will use to optimize the model:
      optimizer = optimizer_init_fn()
    - num_epochs: The number of epochs to train for

    Returns: Nothing, but prints progress during trainingn
    """
    with tf.device(device):
```

```
        # Compute the loss like we did in Part II
        loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()

        model = model_init_fn()
        optimizer = optimizer_init_fn()

        train_loss = tf.keras.metrics.Mean(name='train_loss')
        train_accuracy = tf.keras.metrics.
↪SparseCategoricalAccuracy(name='train_accuracy')

        val_loss = tf.keras.metrics.Mean(name='val_loss')
        val_accuracy = tf.keras.metrics.
↪SparseCategoricalAccuracy(name='val_accuracy')

        t = 0
        for epoch in range(num_epochs):

            # Reset the metrics - https://www.tensorflow.org/alpha/guide/
↪migration_guide#new-style_metrics
            train_loss.reset_states()
            train_accuracy.reset_states()

            for x_np, y_np in train_dset:
                with tf.GradientTape() as tape:

                    # Use the model function to build the forward pass.
                    scores = model(x_np, training=is_training)
                    loss = loss_fn(y_np, scores)

                    gradients = tape.gradient(loss, model.trainable_variables)
                    optimizer.apply_gradients(zip(gradients, model.
↪trainable_variables))

                    # Update the metrics
                    train_loss.update_state(loss)
                    train_accuracy.update_state(y_np, scores)

                    if t % print_every == 0:
                        val_loss.reset_states()
                        val_accuracy.reset_states()
                        for test_x, test_y in val_dset:
                            # During validation at end of epoch, training set␣
↪to False
                            prediction = model(test_x, training=False)
                            t_loss = loss_fn(test_y, prediction)
```

```
                            val_loss.update_state(t_loss)
                            val_accuracy.update_state(test_y, prediction)

                        template = 'Iteration {}, Epoch {}, Loss: {}, Accuracy:␣
↪{}, Val Loss: {}, Val Accuracy: {}'
                        print (template.format(t, epoch+1,
                                             train_loss.result(),
                                             train_accuracy.result()*100,
                                             val_loss.result(),
                                             val_accuracy.result()*100))
                    t += 1
```

6.0.4 Keras Model Subclassing API: Train a Two-Layer Network

We can now use the tools defined above to train a two-layer network on CIFAR-10. We define the `model_init_fn` and `optimizer_init_fn` that construct the model and optimizer respectively when called. Here we want to train the model using stochastic gradient descent with no momentum, so we construct a `tf.keras.optimizers.SGD` function; you can read about it here.

You don't need to tune any hyperparameters here, but you should achieve validation accuracies above 40% after one epoch of training.

[78]:
```
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn():
    return TwoLayerFC(hidden_size, num_classes)

def optimizer_init_fn():
    return tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 3.280930757522583, Accuracy: 4.6875, Val Loss:
2.960726737976074, Val Accuracy: 11.200000762939453
Iteration 100, Epoch 1, Loss: 2.235887050628662, Accuracy: 28.821165084838867,
Val Loss: 1.9042460918426514, Val Accuracy: 37.29999923706055
Iteration 200, Epoch 1, Loss: 2.0729165077209473, Accuracy: 32.72698974609375,
Val Loss: 1.8608322143554688, Val Accuracy: 38.29999923706055
Iteration 300, Epoch 1, Loss: 1.9997899532318115, Accuracy: 34.265987396240234,
Val Loss: 1.9323219060897827, Val Accuracy: 36.39999771118164
Iteration 400, Epoch 1, Loss: 1.9301398992538452, Accuracy: 35.96867370605469,
Val Loss: 1.7239482402801514, Val Accuracy: 41.900001525878906
Iteration 500, Epoch 1, Loss: 1.8884062767028809, Accuracy: 37.072731018066406,
Val Loss: 1.6635971069335938, Val Accuracy: 42.5
Iteration 600, Epoch 1, Loss: 1.8569096326828003, Accuracy: 37.93157196044922,
Val Loss: 1.691287636756897, Val Accuracy: 42.599998474121094
```

```
Iteration 700, Epoch 1, Loss: 1.830712914466858, Accuracy: 38.63453674316406,
Val Loss: 1.6414926052093506, Val Accuracy: 45.0
```

6.0.5 Keras Model Subclassing API: Train a Three-Layer ConvNet

Here you should use the tools we've defined above to train a three-layer ConvNet on CIFAR-10. Your ConvNet should use 32 filters in the first convolutional layer and 16 filters in the second layer.

To train the model you should use gradient descent with Nesterov momentum 0.9.

HINT: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/optimizers/SGD

You don't need to perform any hyperparameter tuning, but you should achieve validation accuracies above 50% after training for one epoch.

[79]:
```
learning_rate = 3e-3
channel_1, channel_2, num_classes = 32, 16, 10

def model_init_fn():
    model = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #Invoking our 3layer Conv class
    model=ThreeLayerConvNet(channel_1,channel_2,num_classes)

    pass

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

def optimizer_init_fn():
    optimizer = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    optimizer=tf.keras.optimizers.SGD(learning_rate=learning_rate,momentum=0.9,␣
↪nesterov=True, name='SGD')

    pass

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

```
    #                              END OF YOUR CODE                              #
    ##############################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.353994846343994, Accuracy: 7.8125, Val Loss:
2.3195013999938965, Val Accuracy: 11.399999618530273
Iteration 100, Epoch 1, Loss: 1.859152913093567, Accuracy: 33.694305419921875,
Val Loss: 1.6237742900848389, Val Accuracy: 43.39999771118164
Iteration 200, Epoch 1, Loss: 1.7090102434158325, Accuracy: 39.38121795654297,
Val Loss: 1.4501216411590576, Val Accuracy: 49.70000076293945
Iteration 300, Epoch 1, Loss: 1.6250770092010498, Accuracy: 42.239410400390625,
Val Loss: 1.4339405298233032, Val Accuracy: 49.0
Iteration 400, Epoch 1, Loss: 1.5607240200042725, Accuracy: 44.65399169921875,
Val Loss: 1.3551909923553467, Val Accuracy: 51.599998474121094
Iteration 500, Epoch 1, Loss: 1.516210675239563, Accuracy: 46.2107048034668, Val
Loss: 1.2972606420516968, Val Accuracy: 54.000003814697266
Iteration 600, Epoch 1, Loss: 1.4886951446533203, Accuracy: 47.21297836303711,
Val Loss: 1.296756625175476, Val Accuracy: 53.89999771118164
Iteration 700, Epoch 1, Loss: 1.4623479843139648, Accuracy: 48.18117141723633,
Val Loss: 1.2593278884887695, Val Accuracy: 56.599998474121094
```

7 Part IV: Keras Sequential API

In Part III we introduced the `tf.keras.Model` API, which allows you to define models with any number of learnable layers and with arbitrary connectivity between layers.

However for many models you don't need such flexibility - a lot of models can be expressed as a sequential stack of layers, with the output of each layer fed to the next layer as input. If your model fits this pattern, then there is an even easier way to define your model: using `tf.keras.Sequential`. You don't need to write any custom classes; you simply call the `tf.keras.Sequential` constructor with a list containing a sequence of layer objects.

One complication with `tf.keras.Sequential` is that you must define the shape of the input to the model by passing a value to the `input_shape` of the first layer in your model.

7.0.1 Keras Sequential API: Two-Layer Network

In this subsection, we will rewrite the two-layer fully-connected network using `tf.keras.Sequential`, and train it using the training loop defined above.

You don't need to perform any hyperparameter tuning here, but you should see validation accuracies above 40% after training for one epoch.

[80]:
```
learning_rate = 1e-2
```

```
def model_init_fn():
    input_shape = (32, 32, 3)
    hidden_layer_size, num_classes = 4000, 10
    initializer = tf.initializers.VarianceScaling(scale=2.0)
    layers = [
        tf.keras.layers.Flatten(input_shape=input_shape),
        tf.keras.layers.Dense(hidden_layer_size, activation='relu',
                              kernel_initializer=initializer),
        tf.keras.layers.Dense(num_classes, activation='softmax',
                              kernel_initializer=initializer),
    ]
    model = tf.keras.Sequential(layers)
    return model

def optimizer_init_fn():
    return tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.6873698234558105, Accuracy: 10.9375, Val Loss:
2.704998731613159, Val Accuracy: 15.899999618530273
Iteration 100, Epoch 1, Loss: 2.222005844116211, Accuracy: 28.589109420776367,
Val Loss: 1.8801274299621582, Val Accuracy: 37.70000076293945
Iteration 200, Epoch 1, Loss: 2.068542242050171, Accuracy: 32.68034744262695,
Val Loss: 1.83555006980896, Val Accuracy: 38.5
Iteration 300, Epoch 1, Loss: 1.9960688352584839, Accuracy: 34.54111099243164,
Val Loss: 1.904266357421875, Val Accuracy: 36.099998474121094
Iteration 400, Epoch 1, Loss: 1.9311195611953735, Accuracy: 36.24532699584961,
Val Loss: 1.708775281906128, Val Accuracy: 41.70000076293945
Iteration 500, Epoch 1, Loss: 1.887345552444458, Accuracy: 37.24114227294922,
Val Loss: 1.6509883403778076, Val Accuracy: 42.79999923706055
Iteration 600, Epoch 1, Loss: 1.857306957244873, Accuracy: 38.17335510253906,
Val Loss: 1.6876699924468994, Val Accuracy: 41.20000076293945
Iteration 700, Epoch 1, Loss: 1.8313366174697876, Accuracy: 38.7526741027832,
Val Loss: 1.6217623949050903, Val Accuracy: 45.10000228881836
```

7.0.2 Abstracting Away the Training Loop

In the previous examples, we used a customised training loop to train models (e.g. `train_part34`). Writing your own training loop is only required if you need more flexibility and control during training your model. Alternately, you can also use built-in APIs like `tf.keras.Model.fit()` and `tf.keras.Model.evaluate` to train and evaluate a model. Also remember to configure your model for training by calling ‘tf.keras.Model.compile.

You don’t need to perform any hyperparameter tuning here, but you should see validation and test accuracies above 42% after training for one epoch.

[81]:
```
model = model_init_fn()
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=learning_rate),
              loss='sparse_categorical_crossentropy',
              metrics=[tf.keras.metrics.sparse_categorical_accuracy])
model.fit(X_train, y_train, batch_size=64, epochs=1, validation_data=(X_val,␣
 ↪y_val))
model.evaluate(X_test, y_test)
```

```
766/766 [==============================] - 4s 4ms/step - loss: 1.8231 -
sparse_categorical_accuracy: 0.3889 - val_loss: 1.6779 -
val_sparse_categorical_accuracy: 0.4520
313/313 [==============================] - 1s 3ms/step - loss: 1.6503 -
sparse_categorical_accuracy: 0.4367
```

[81]:
```
[1.6503291130065918, 0.4366999864578247]
```

7.0.3 Keras Sequential API: Three-Layer ConvNet

Here you should use `tf.keras.Sequential` to reimplement the same three-layer ConvNet architecture used in Part II and Part III. As a reminder, your model should have the following architecture:

1. Convolutional layer with 32 5x5 kernels, using zero padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 16 3x3 kernels, using zero padding of 1
4. ReLU nonlinearity
5. Fully-connected layer giving class scores
6. Softmax nonlinearity

You should initialize the weights of the model using a `tf.initializers.VarianceScaling` as above.

You should train the model using Nesterov momentum 0.9.

You don't need to perform any hyperparameter search, but you should achieve accuracy above 45% after training for one epoch.

[87]:
```
def model_init_fn():
    model = None
    ##############################################################################
    # TODO: Construct a three-layer ConvNet using tf.keras.Sequential.           #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    channel_1, channel_2, num_classes = 32, 16, 10
    initializer = tf.initializers.VarianceScaling(scale=2.0)
    #Defining filter and padding
    filter1=(5,5)
    filter2=(3,3)
    #Initializing sequential layers as we did in part II
```

```
    layers=[
        tf.keras.layers.ZeroPadding2D(padding=(2,2)),
        tf.keras.layers.ZeroPadding2D(padding=(1,1)),
        tf.keras.layers.Conv2D(kernel_size =filter1, filters = channel_1,
                                             padding = 'valid',␣
↪activation='relu'),
        tf.keras.layers.Conv2D(kernel_size =filter2, filters = channel_2,
                                             padding = 'valid',␣
↪activation='relu'),
        tf.keras.layers.Flatten(),

        tf.keras.layers.Dense(num_classes, activation='softmax',
                                   kernel_initializer=initializer)
    ]
    model=tf.keras.Sequential(layers)

    pass

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

learning_rate = 5e-4
def optimizer_init_fn():
    optimizer = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #Using the same optmizier value as we did in Part III for this sequential␣
↪process
    optimizer=tf.keras.optimizers.SGD(learning_rate=learning_rate,momentum=0.9,␣
↪nesterov=True, name='SGD')


    pass

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

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.336310863494873, Accuracy: 9.375, Val Loss:
2.355645179748535, Val Accuracy: 7.90000057220459
Iteration 100, Epoch 1, Loss: 2.134108781814575, Accuracy: 22.38551902770996,
Val Loss: 2.0002479553222656, Val Accuracy: 29.19999885559082
Iteration 200, Epoch 1, Loss: 2.041557550430298, Accuracy: 26.826805114746094,
Val Loss: 1.8702236413955688, Val Accuracy: 36.70000076293945
Iteration 300, Epoch 1, Loss: 1.978104591369629, Accuracy: 29.749792098999023,
Val Loss: 1.7851388454437256, Val Accuracy: 42.099998474121094
Iteration 400, Epoch 1, Loss: 1.9135240316390991, Accuracy: 32.228023529052734,
Val Loss: 1.7115960121154785, Val Accuracy: 42.39999771118164
Iteration 500, Epoch 1, Loss: 1.8667300939559937, Accuracy: 33.972679138183594,
Val Loss: 1.6436198949813843, Val Accuracy: 42.0
Iteration 600, Epoch 1, Loss: 1.8300198316574097, Accuracy: 35.40193557739258,
Val Loss: 1.6096521615982056, Val Accuracy: 43.900001525878906
Iteration 700, Epoch 1, Loss: 1.7972588539123535, Accuracy: 36.66191101074219,
Val Loss: 1.5602717399597168, Val Accuracy: 45.20000076293945
```

We will also train this model with the built-in training loop APIs provided by TensorFlow.

[88]:
```
model = model_init_fn()
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=[tf.keras.metrics.sparse_categorical_accuracy])
model.fit(X_train, y_train, batch_size=64, epochs=1, validation_data=(X_val,␣
 ↪y_val))
model.evaluate(X_test, y_test)
```

```
766/766 [==============================] - 5s 6ms/step - loss: 1.6176 -
sparse_categorical_accuracy: 0.4239 - val_loss: 1.4662 -
val_sparse_categorical_accuracy: 0.4670
313/313 [==============================] - 1s 3ms/step - loss: 1.4662 -
sparse_categorical_accuracy: 0.4765
```

[88]:
```
[1.4661531448364258, 0.476500004529953]
```

7.1 Part IV: Functional API

7.1.1 Demonstration with a Two-Layer Network

In the previous section, we saw how we can use `tf.keras.Sequential` to stack layers to quickly build simple models. But this comes at the cost of losing flexibility.

Often we will have to write complex models that have non-sequential data flows: a layer can have **multiple inputs and/or outputs**, such as stacking the output of 2 previous layers together to feed as input to a third! (Some examples are residual connections and dense blocks.)

In such cases, we can use Keras functional API to write models with complex topologies such as:

1. Multi-input models

2. Multi-output models
3. Models with shared layers (the same layer called several times)
4. Models with non-sequential data flows (e.g. residual connections)

Writing a model with Functional API requires us to create a `tf.keras.Model` instance and explicitly write input tensors and output tensors for this model.

[89]:
```
def two_layer_fc_functional(input_shape, hidden_size, num_classes):
    initializer = tf.initializers.VarianceScaling(scale=2.0)
    inputs = tf.keras.Input(shape=input_shape)
    flattened_inputs = tf.keras.layers.Flatten()(inputs)
    fc1_output = tf.keras.layers.Dense(hidden_size, activation='relu',
                                                                      ␣
 ↪kernel_initializer=initializer)(flattened_inputs)
    scores = tf.keras.layers.Dense(num_classes, activation='softmax',
                                   kernel_initializer=initializer)(fc1_output)

    # Instantiate the model given inputs and outputs.
    model = tf.keras.Model(inputs=inputs, outputs=scores)
    return model

def test_two_layer_fc_functional():
    """ A small unit test to exercise the TwoLayerFC model above. """
    input_size, hidden_size, num_classes = 50, 42, 10
    input_shape = (50,)

    x = tf.zeros((64, input_size))
    model = two_layer_fc_functional(input_shape, hidden_size, num_classes)

    with tf.device(device):
        scores = model(x)
        print(scores.shape)

test_two_layer_fc_functional()
```

```
(64, 10)
```

7.1.2 Keras Functional API: Train a Two-Layer Network

You can now train this two-layer network constructed using the functional API.

You don't need to perform any hyperparameter tuning here, but you should see validation accuracies above 40% after training for one epoch.

[90]:
```
input_shape = (32, 32, 3)
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2
```

```python
def model_init_fn():
    return two_layer_fc_functional(input_shape, hidden_size, num_classes)

def optimizer_init_fn():
    return tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.972186803817749, Accuracy: 12.5, Val Loss:
2.6536083221435547, Val Accuracy: 13.300000190734863
Iteration 100, Epoch 1, Loss: 2.2487902641296387, Accuracy: 28.27970314025879,
Val Loss: 1.8742800951004028, Val Accuracy: 37.400001525878906
Iteration 200, Epoch 1, Loss: 2.088994026184082, Accuracy: 31.763059616088867,
Val Loss: 1.7872952222824097, Val Accuracy: 41.900001525878906
Iteration 300, Epoch 1, Loss: 2.007378101348877, Accuracy: 33.82994079589844,
Val Loss: 1.8893846273422241, Val Accuracy: 38.0
Iteration 400, Epoch 1, Loss: 1.9340589046478271, Accuracy: 35.85956954956055,
Val Loss: 1.6969997882843018, Val Accuracy: 43.29999923706055
Iteration 500, Epoch 1, Loss: 1.8893204927444458, Accuracy: 37.00411605834961,
Val Loss: 1.6612651348114014, Val Accuracy: 42.89999771118164
Iteration 600, Epoch 1, Loss: 1.857300877571106, Accuracy: 37.98357009887695,
Val Loss: 1.67881441116333, Val Accuracy: 41.29999923706055
Iteration 700, Epoch 1, Loss: 1.8314919471740723, Accuracy: 38.661285400390625,
Val Loss: 1.6177536249160767, Val Accuracy: 44.10000228881836
```

8 Part V: CIFAR-10 open-ended challenge

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

You should experiment with architectures, hyperparameters, loss functions, regularization, or anything else you can think of to train a model that achieves **at least 70%** accuracy on the **validation** set within 10 epochs. You can use the built-in train function, the `train_part34` function from above, or implement your own training loop.

Describe what you did at the end of the notebook.

8.0.1 Some things you can try:

- **Filter size**: Above we used 5x5 and 3x3; is this optimal?
- **Number of filters**: Above we used 16 and 32 filters. Would more or fewer do better?
- **Pooling**: We didn't use any pooling above. Would this improve the model?
- **Normalization**: Would your model be improved with batch normalization, layer normalization, group normalization, or some other normalization strategy?
- **Network architecture**: The ConvNet above has only three layers of trainable parameters. Would a deeper model do better?

- **Global average pooling**: Instead of flattening after the final convolutional layer, would global average pooling do better? This strategy is used for example in Google's Inception network and in Residual Networks.
- **Regularization**: Would some kind of regularization improve performance? Maybe weight decay or dropout?

8.0.2 NOTE: Batch Normalization / Dropout

If you are using Batch Normalization and Dropout, remember to pass `is_training=True` if you use the `train_part34()` function. BatchNorm and Dropout layers have different behaviors at training and inference time. `training` is a specific keyword argument reserved for this purpose in any `tf.keras.Model`'s `call()` function. Read more about this here : https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/BatchNormalization#methods https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Dropout#methods

8.0.3 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.

8.0.4 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

8.0.5 Have fun and happy training!

[112]:
```
class CustomConvNet(tf.keras.Model):
    def __init__(self):
        super(CustomConvNet, self).__init__()
        ␣
↪############################################################################
        # TODO: Construct a model that performs well on CIFAR-10               ␣
↪   #
        ␣
↪############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.padding1=(2,2)
        self.padding2=(1,1)
        self.padding3=(1,1)
        self.filter1=(5,5)
        self.filter2=(4,4)
        self.filter3=(3,3)
        #Here we increase channel sizes from 32 and 16 to 32 and 128
        channel_1,channel_2,channel_3,num_classes=32,64,128,10 #CIFAR has 10␣
↪classes
        #Intializer for FC Net (As done in 2 FC net)
        initializer = tf.initializers.VarianceScaling(scale=2.0)
        #Initialize Zero Padding using https://www.tensorflow.org/api_docs/
↪python/tf/keras/layers/ZeroPadding2D4
        self.padding1=tf.keras.layers.ZeroPadding2D(padding=self.padding1)
        self.padding2=tf.keras.layers.ZeroPadding2D(padding=self.padding2)
        self.padding3=tf.keras.layers.ZeroPadding2D(padding=self.padding3)
        #Performing convilution for for the first kayer using valid padding as␣
↪mentioned
        self.conv1=tf.keras.layers.Conv2D(kernel_size = self.filter1, filters =␣
↪channel_1,
                                           padding = 'valid',␣
↪activation='relu')
        #Performing MaxPool for a standard (2,2) MaxPool Layer1
        self.pool1=tf.keras.layers.MaxPool2D(pool_size=(2,2))
        #Performing BN for layer 1
        self.bn1=tf.keras.layers.BatchNormalization()
        #Performing convilution for for the second layer using valid padding as␣
↪mentioned
        self.conv2=tf.keras.layers.Conv2D(kernel_size = self.filter1, filters =␣
↪channel_2,
                                           padding = 'valid',␣
↪activation='relu')
        #Performing MaxPool for a standard (2,2) MaxPool Layer2
        self.pool2=tf.keras.layers.MaxPool2D(pool_size=(2,2))
```

```
        #Performing BatchNormalization
        self.bn2=tf.keras.layers.BatchNormalization()
        #Performing convilution for for the third kayer using valid padding as␣
↪mentioned
        self.conv3=tf.keras.layers.Conv2D(kernel_size = self.filter1, filters =␣
↪channel_3,
                                            padding = 'valid',␣
↪activation='relu')
        #Performing MaxPool for a standard (2,2) MaxPool Layer1
        self.pool3=tf.keras.layers.MaxPool2D(pool_size=(2,2))
        #Performing BN for layer 1
        self.bn3=tf.keras.layers.BatchNormalization()


        #flattening the conv output
        self.flatten = tf.keras.layers.Flatten()
        self.fc = tf.keras.layers.Dense(num_classes, activation='softmax',
                                    kernel_initializer=initializer)
        #adding dropout for the FC Layer
        self.drop1 = tf.keras.layers.Dropout(0.2)
        self.fc2 = tf.keras.layers.Dense(num_classes, activation='softmax',
                                    kernel_initializer=initializer)

        pass


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

    def call(self, input_tensor, training=False):
      ␣
↪############################################################################
        # TODO: Construct a model that performs well on CIFAR-10             ␣
↪   #
      ␣
↪############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        #Calling back the class objects using x for first layer
        x = self.padding1(input_tensor)
        x = self.conv1(x)
       #Calling BN and MaxPool
```

```
        x = self.bn1(x)
        x = self.pool1(x)
       #Calling back the class objects using x for second layer as its forward␣
 ↪continous pass
        x = self.padding2(x)
        x = self.conv2(x)
      #Calling BN and MaxPool
        x = self.bn2(x)
        x = self.pool2(x)
    #Calling back the class objects using x for third layer as its forward␣
 ↪continous pass
        x = self.padding3(x)
        x = self.conv3(x)
      #Calling BN and MaxPool
        x = self.bn3(x)
        x = self.pool3(x)


      #Calling back the class objects using x for flattening layer
        x = self.flatten(x)
        #The final layer score is computed after the fc net which is then␣
 ↪stored as scores
        x = self.fc(x)
        x= self.drop1(x)
        x=self.fc2(x)


        pass

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


print_every = 700
num_epochs = 10

model = CustomConvNet()

def model_init_fn():
    return CustomConvNet()
```

```python
def optimizer_init_fn():
    learning_rate = 1e-3
    return tf.keras.optimizers.Adam(learning_rate)

train_part34(model_init_fn, optimizer_init_fn, num_epochs=num_epochs,␣
↪is_training=True)
```

```
Iteration 0, Epoch 1, Loss: 2.4102025032043457, Accuracy: 10.9375, Val Loss:
2.321512460708618, Val Accuracy: 7.800000190734863
Iteration 700, Epoch 1, Loss: 1.9772181510925293, Accuracy: 30.895597457885742,
Val Loss: 1.751899003982544, Val Accuracy: 45.400001525878906
Iteration 1400, Epoch 2, Loss: 1.7092223167419434, Accuracy: 39.156005859375,
Val Loss: 1.5612232685089111, Val Accuracy: 48.79999923706055
Iteration 2100, Epoch 3, Loss: 1.5346790552139282, Accuracy: 45.68046951293945,
Val Loss: 1.4054561853408813, Val Accuracy: 52.60000228881836
Iteration 2800, Epoch 4, Loss: 1.3958176374435425, Accuracy: 49.69247055053711,
Val Loss: 1.2345821857452393, Val Accuracy: 58.29999923706055
Iteration 3500, Epoch 5, Loss: 1.2962591648101807, Accuracy: 52.30978012084961,
Val Loss: 1.2038666009902954, Val Accuracy: 57.599998474121094
Iteration 4200, Epoch 6, Loss: 1.217015027999878, Accuracy: 54.3421516418457,
Val Loss: 1.0607565641403198, Val Accuracy: 63.599998474121094
Iteration 4900, Epoch 7, Loss: 1.1420886516571045, Accuracy: 58.360652923583984,
Val Loss: 1.0600168704986572, Val Accuracy: 64.20000457763672
Iteration 5600, Epoch 8, Loss: 1.0591108798980713, Accuracy: 62.330020904541016,
Val Loss: 1.0031834840774536, Val Accuracy: 65.69999694824219
Iteration 6300, Epoch 9, Loss: 0.9832340478897095, Accuracy: 67.3320083618164,
Val Loss: 1.050566554069519, Val Accuracy: 68.5
Iteration 7000, Epoch 10, Loss: 0.9068604111671448, Accuracy: 69.20268249511719,
Val Loss: 0.9677135348320007, Val Accuracy: 70.9000015258789
```

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:

Architecture : [Conv -> RELU -> BN -> Pool] x3 -> FC1 -> Drop -> Fc2

Filters : * 5,5 for Conv1 * 4,4 for Conv2
* 3,3 for conv3 channel size = 32,64,128 for conv 1 conv2 conv3 respectively

- Can notice that for smaller sizes the accuracy was lesser so decided to increase size of channels.

MaxPool size used = standard 2x2

Dropout - Rate =0.2 [**lower the dropout rate, lower the loss , higher the accuracy**]

- After the first layer, the filter size is reduced , which brings us higher accuracy and lesser loss in second epoch.
- Batchnorm was used after every conv layer as it enables in training large datasets and brings to faster convergance.
- Dropout was used to prevent model overfitting. Higher the dropout rate , more the chances of model not fitting properly.