Q1-FCNetworks

October 5, 2025

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

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'ece697ls/assignments/assignment3/'
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))

%cd /content
```

1 Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks using a modular approach. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

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

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

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients

with respect to the inputs and weights, like this:

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

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

  return dx, dw
```

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

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

[2]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from ece662.classifiers.fc_net import *
from ece662.data_utils import get_CINIC10_data
from ece662.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from ece662.solver import Solver

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

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

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

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

[3]:
```
# Load the (preprocessed) CINIC10 data - Note that CINIC10 was modified in size␣
 ↪for this course

data = get_CINIC10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))

print('Number of Classes: {}'.format(len(np.unique(data['y_train']))))
```

```
('X_train: ', (53973, 3, 32, 32))
('y_train: ', (53973,))
('X_val: ', (10195, 3, 32, 32))
('y_val: ', (10195,))
('X_test: ', (10196, 3, 32, 32))
('y_test: ', (10196,))
Number of Classes: 6
```

2 Affine layer: forward

Open the file `ece697ls/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

[4]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

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

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),␣
 ↪output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
```

```
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

3 Affine layer: backward

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

[5]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

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

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
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 affine_backward function:
dx error:  9.579607670311995e-11
dw error:  1.2604803862099753e-10
db error:  7.72167374950876e-11
```

4 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

[6]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

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

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

[7]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

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

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. Tanh

5.2 Answer:

Both Sigmoid and Tanh can suffer from getting zero gradients during backpropagation.

We have Sigmoid(x) = 1/(1+e^(-x)) -> derivative of it is: Sigmoid(x)*(1-Sigmoid(x)). We have for very large x, both positive and negative, the derivative of Sigmoid(x) -> 0 as Sigmoid -> 1. Thus Sigmoid would suffer from zero gradients.

The same apply to Tanh. Tanh(x) = (e^x - e^(-x))/(e^x + e^(-x)) and the derivative is 1 - Tanh^2(x). Again for very large x, both positive and negative, the derivative of Tanh(x) -> 0 as Tanh(x) -> 1.

5.3 Inline Question 2:

Why is an activation function required ? Why is a bias required ?

5.4 Answer:

5.4.1 Activation function

If we didn't have non-linear activations, a neural network would just be a bunch of linear transformations stacked together, only represent linear relationship. That means the network wouldn't be able to model anything beyond straight-line relationships. By adding activation functions, we inject non-linearity, which is what lets the network actually capture and learn more complex patterns.

5.4.2 Bias

Bias is basically a way of shifting the activation function left or right. Without it, every function the network learns would have to pass through the origin, constraining the model.

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `ece697ls/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

[8]:
```
from ece662.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

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

```
# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
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 affine_relu_forward and affine_relu_backward:
dx error:  1.70644420803027e-10
dw error:  8.162088973990222e-11
db error:  7.82672402145899e-12
```

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `ece697ls/layers.py`.

You can make sure that the implementations are correct by running the following:

[9]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

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

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x,␣
 ↪verbose=False)
loss, dx = softmax_loss(x, y)

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

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09
```

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

8 Two-layer network

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

Open the file `ece697ls/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

[13]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
```

```
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-08
W2 relative error: 3.77e-10
b1 relative error: 9.83e-09
b2 relative error: 1.94e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 1.37e-07
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

9 Solver

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

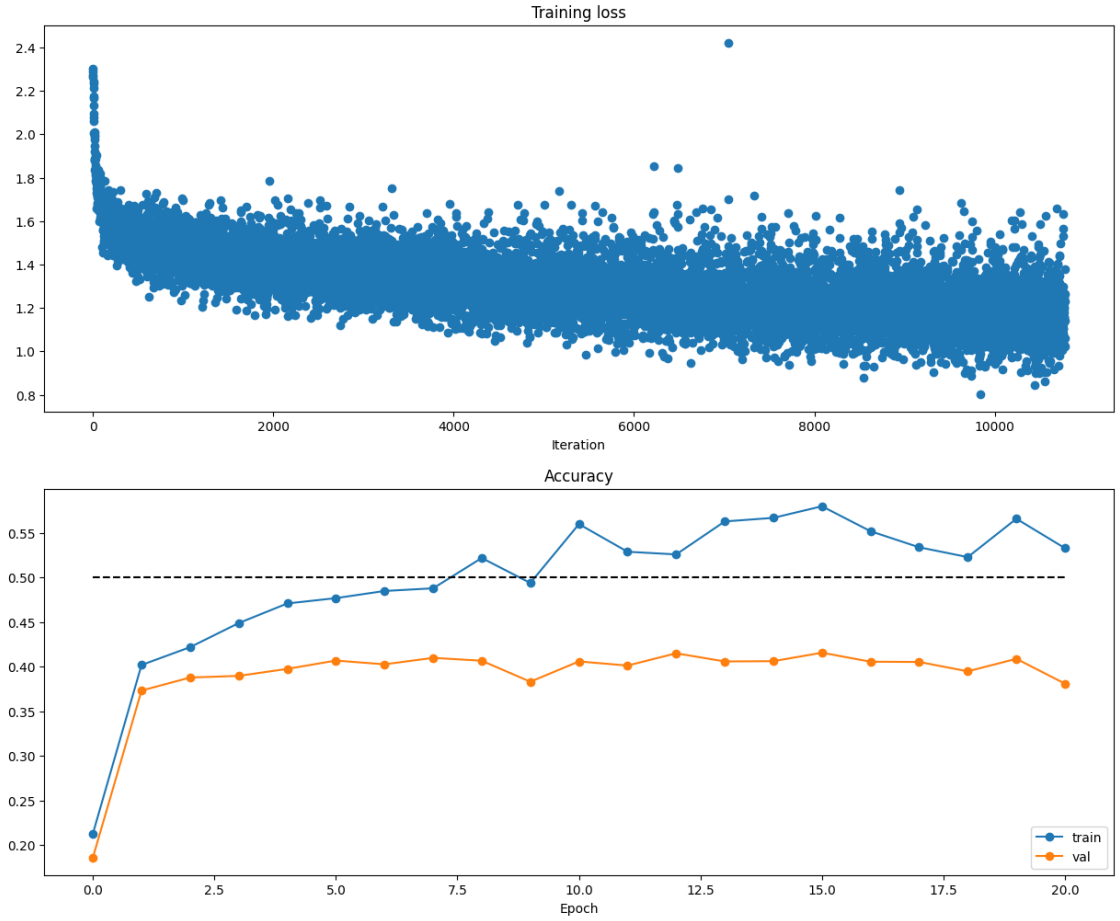
Open the file `ece6971s/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `40%` accuracy on the validation set.

[15]:
```
model = TwoLayerNet()
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                          #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

solver = Solver(model, data,
                update_rule='sgd',
                optim_config={'learning_rate': 1e-3},
                batch_size=100,
                num_epochs=20,
                print_every=1000)
solver.train()

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

```
(Iteration 1 / 10780) loss: 2.308233
(Epoch 0 / 20) train acc: 0.174000; val_acc: 0.155665
(Epoch 1 / 20) train acc: 0.404000; val_acc: 0.381265
(Iteration 1001 / 10780) loss: 1.412765
(Epoch 2 / 20) train acc: 0.421000; val_acc: 0.386366
(Epoch 3 / 20) train acc: 0.449000; val_acc: 0.407160
(Iteration 2001 / 10780) loss: 1.355938
(Epoch 4 / 20) train acc: 0.447000; val_acc: 0.394213
(Epoch 5 / 20) train acc: 0.517000; val_acc: 0.410888
(Iteration 3001 / 10780) loss: 1.338313
(Epoch 6 / 20) train acc: 0.476000; val_acc: 0.408239
(Epoch 7 / 20) train acc: 0.494000; val_acc: 0.401569
(Iteration 4001 / 10780) loss: 1.338962
(Epoch 8 / 20) train acc: 0.492000; val_acc: 0.404120
(Epoch 9 / 20) train acc: 0.504000; val_acc: 0.403041
(Iteration 5001 / 10780) loss: 1.136491
(Epoch 10 / 20) train acc: 0.480000; val_acc: 0.398921
(Epoch 11 / 20) train acc: 0.556000; val_acc: 0.414223
(Iteration 6001 / 10780) loss: 1.363951
(Epoch 12 / 20) train acc: 0.521000; val_acc: 0.409514
(Iteration 7001 / 10780) loss: 1.201890
(Epoch 13 / 20) train acc: 0.493000; val_acc: 0.392643
(Epoch 14 / 20) train acc: 0.539000; val_acc: 0.401373
(Iteration 8001 / 10780) loss: 1.190673
(Epoch 15 / 20) train acc: 0.549000; val_acc: 0.403041
(Epoch 16 / 20) train acc: 0.585000; val_acc: 0.391957
```

```
(Iteration 9001 / 10780) loss: 1.251584
(Epoch 17 / 20) train acc: 0.531000; val_acc: 0.384208
(Epoch 18 / 20) train acc: 0.580000; val_acc: 0.410790
(Iteration 10001 / 10780) loss: 1.238688
(Epoch 19 / 20) train acc: 0.590000; val_acc: 0.413046
(Epoch 20 / 20) train acc: 0.582000; val_acc: 0.399706
```

[24]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

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



10 Multilayer network

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

Read through the `FullyConnectedNet` class in the file `ece697ls/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

10.1 Initial loss and gradient check

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

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

[25]:
```
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('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

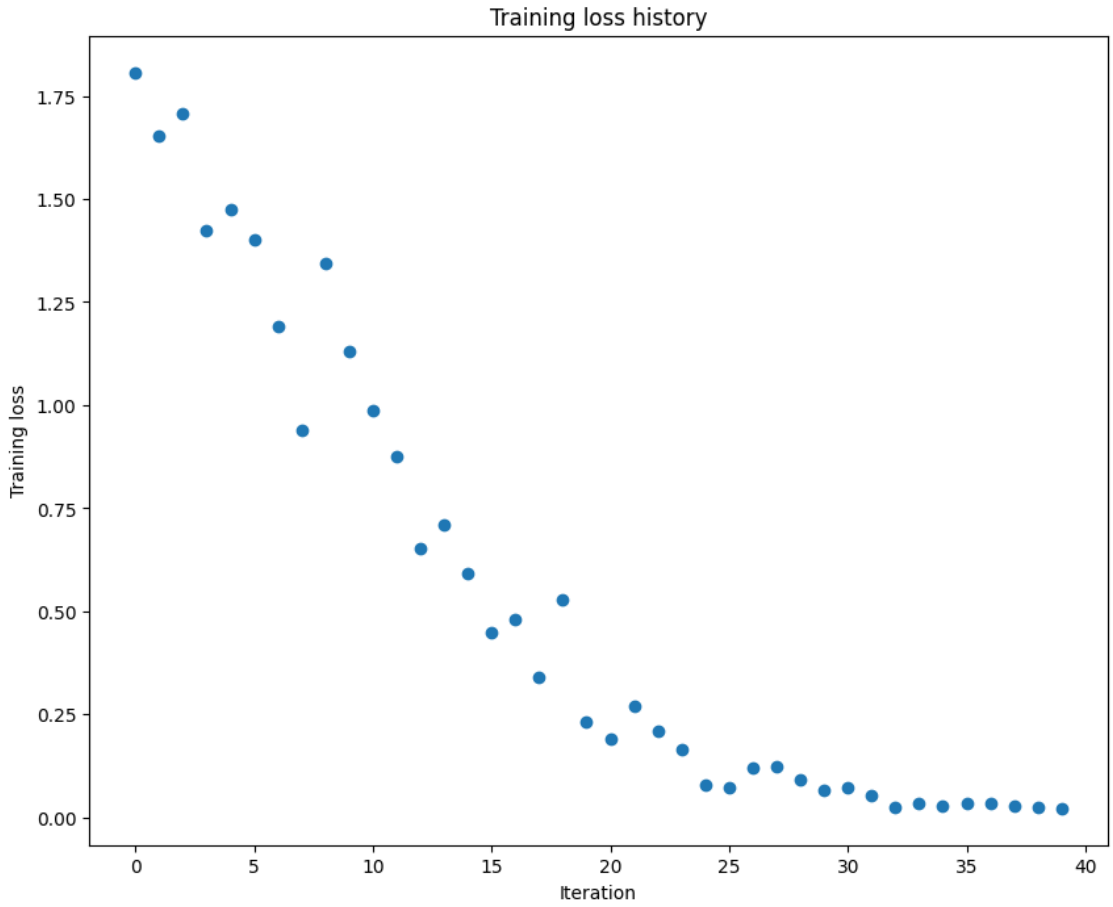
```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 3.90e-09
W2 relative error: 3.52e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 90+% training accuracy within 20 epochs.

[26]:
```
# 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-2  # Experiment with this!
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64, num_classes=6)
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, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 1.806892
(Epoch 0 / 20) train acc: 0.300000; val_acc: 0.198431
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.206376
(Epoch 2 / 20) train acc: 0.360000; val_acc: 0.189308
(Epoch 3 / 20) train acc: 0.600000; val_acc: 0.217656
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.217656
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.186758
(Iteration 11 / 40) loss: 0.986982
(Epoch 6 / 20) train acc: 0.820000; val_acc: 0.232075
(Epoch 7 / 20) train acc: 0.940000; val_acc: 0.216381
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.227857
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.224816
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.233055
(Iteration 21 / 40) loss: 0.191172
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.234723
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.232565
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.227268
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.231878
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.234821
(Iteration 31 / 40) loss: 0.073391
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.236292
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.235802
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.239627
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.236783
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.228347
```

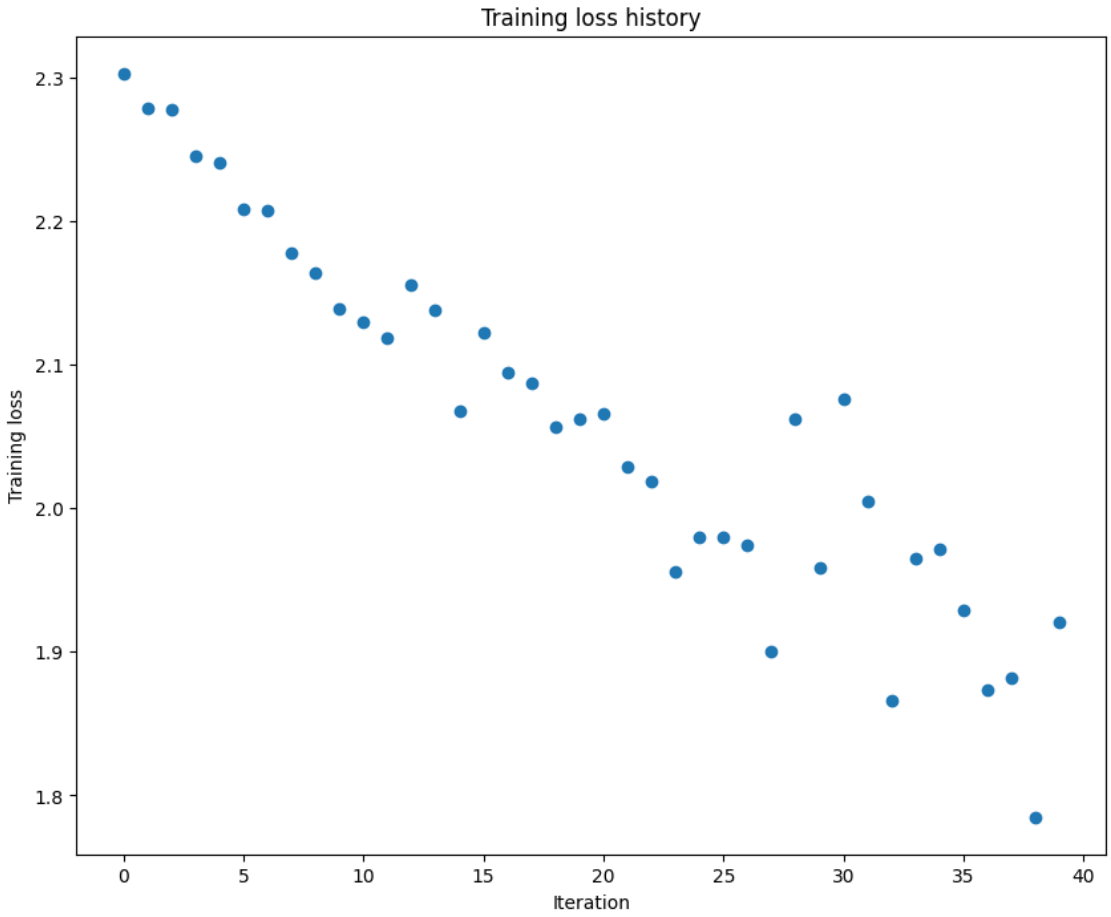


Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[27]:
```
# 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-1  # Experiment with this!
weight_scale = 1e-5   # Experiment with this!
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
```

```
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

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

```
(Iteration 1 / 40) loss: 2.302585
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 1 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 2 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 3 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 4 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 5 / 20) train acc: 0.240000; val_acc: 0.166650
(Iteration 11 / 40) loss: 2.129272
(Epoch 6 / 20) train acc: 0.240000; val_acc: 0.166650
(Epoch 7 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 8 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 9 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 10 / 20) train acc: 0.260000; val_acc: 0.166650
(Iteration 21 / 40) loss: 2.066156
(Epoch 11 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 12 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 13 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 14 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 15 / 20) train acc: 0.260000; val_acc: 0.166650
(Iteration 31 / 40) loss: 2.075461
(Epoch 16 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 17 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 18 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 19 / 20) train acc: 0.260000; val_acc: 0.166650
(Epoch 20 / 20) train acc: 0.260000; val_acc: 0.166650
```



10.2 Inline Question 3:

Why would you need multi-layer networks ? Will stacking several layers without activation functions be able to model complex problems ?

10.3 Answer:

We use multi-layer networks because real-world problems are complex and one layer can't learn that complex pattern.

If we stack just linear layers without any activation functions, the whole networks will, indeed, turn into a single layer, losing the point of having multiple layers networks. Activations are what give those layers real power. Activation functions are what make deeper networks powerful, since they introduce the non-linearity needed to model complex data.

11 Update rules

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

commonly used update rules and compare them to vanilla SGD.

12 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.

Open the file `ece697ls/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.

[28]:
```
from ece662.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.

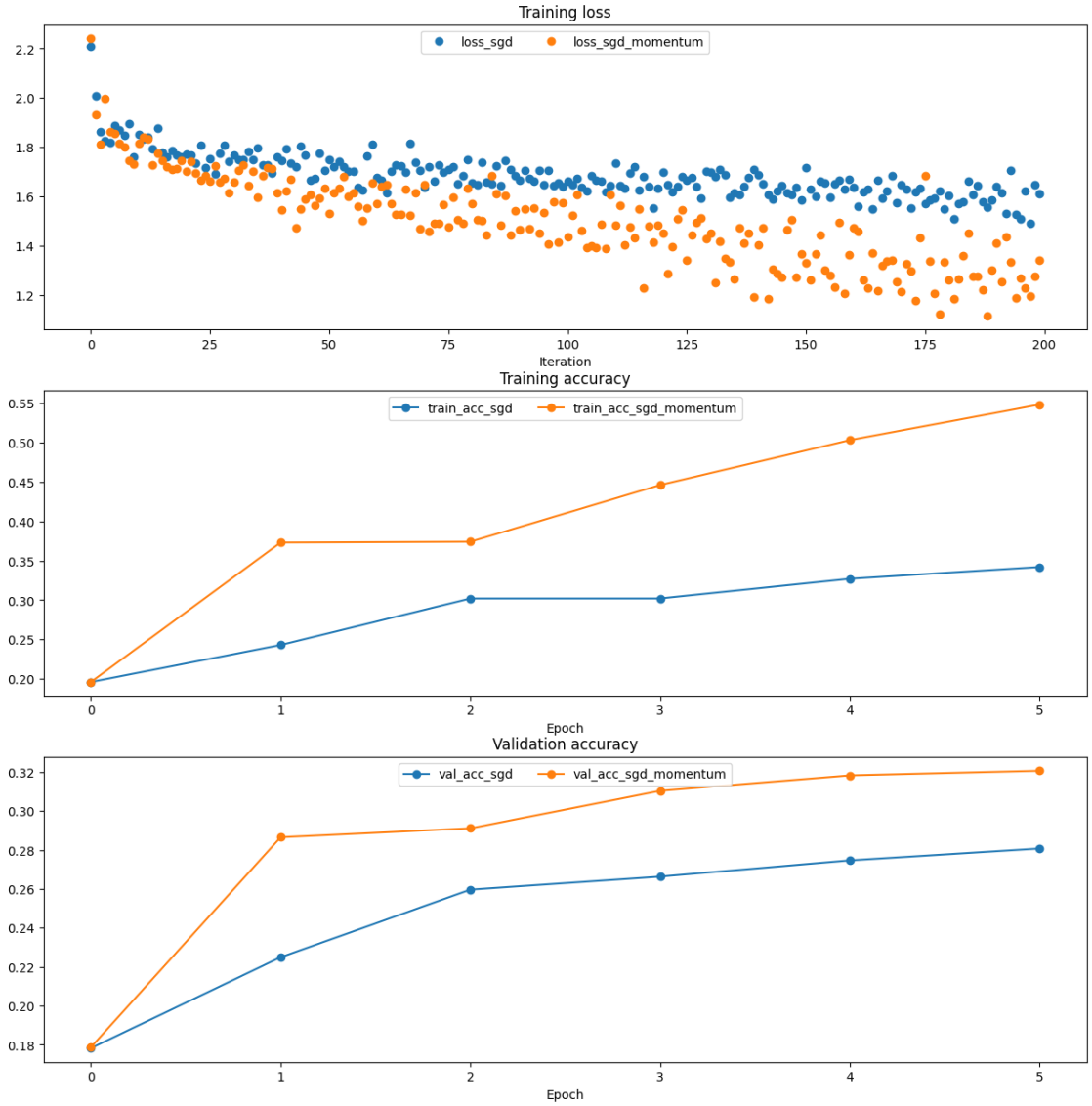
[29]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}
```

```
solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2,␣
 ↪num_classes=6)

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

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

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

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

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

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

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

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

```
running with  sgd
```

```
(Iteration 1 / 200) loss: 2.208851
(Epoch 0 / 5) train acc: 0.196000; val_acc: 0.178225
(Iteration 11 / 200) loss: 1.852902
(Iteration 21 / 200) loss: 1.770052
(Iteration 31 / 200) loss: 1.766842
(Epoch 1 / 5) train acc: 0.243000; val_acc: 0.224914
(Iteration 41 / 200) loss: 1.743974
(Iteration 51 / 200) loss: 1.747804
(Iteration 61 / 200) loss: 1.675048
(Iteration 71 / 200) loss: 1.636662
(Epoch 2 / 5) train acc: 0.302000; val_acc: 0.259637
(Iteration 81 / 200) loss: 1.654430
(Iteration 91 / 200) loss: 1.663805
(Iteration 101 / 200) loss: 1.660197
(Iteration 111 / 200) loss: 1.732837
(Epoch 3 / 5) train acc: 0.302000; val_acc: 0.266307
(Iteration 121 / 200) loss: 1.696295
(Iteration 131 / 200) loss: 1.697090
(Iteration 141 / 200) loss: 1.686636
(Iteration 151 / 200) loss: 1.714582
(Epoch 4 / 5) train acc: 0.327000; val_acc: 0.274644
(Iteration 161 / 200) loss: 1.636100
(Iteration 171 / 200) loss: 1.646538
(Iteration 181 / 200) loss: 1.604620
(Iteration 191 / 200) loss: 1.638678
(Epoch 5 / 5) train acc: 0.342000; val_acc: 0.280726

running with  sgd_momentum
(Iteration 1 / 200) loss: 2.241194
(Epoch 0 / 5) train acc: 0.196000; val_acc: 0.178715
(Iteration 11 / 200) loss: 1.816656
(Iteration 21 / 200) loss: 1.703495
(Iteration 31 / 200) loss: 1.658050
(Epoch 1 / 5) train acc: 0.373000; val_acc: 0.286513
(Iteration 41 / 200) loss: 1.546582
(Iteration 51 / 200) loss: 1.531124
(Iteration 61 / 200) loss: 1.572017
(Iteration 71 / 200) loss: 1.648156
(Epoch 2 / 5) train acc: 0.374000; val_acc: 0.291123
(Iteration 81 / 200) loss: 1.571354
(Iteration 91 / 200) loss: 1.464847
(Iteration 101 / 200) loss: 1.435939
(Iteration 111 / 200) loss: 1.483779
(Epoch 3 / 5) train acc: 0.446000; val_acc: 0.310348
(Iteration 121 / 200) loss: 1.451774
(Iteration 131 / 200) loss: 1.448878
(Iteration 141 / 200) loss: 1.401287
(Iteration 151 / 200) loss: 1.329060
```

```
(Epoch 4 / 5) train acc: 0.503000; val_acc: 0.318293
(Iteration 161 / 200) loss: 1.470376
(Iteration 171 / 200) loss: 1.212996
(Iteration 181 / 200) loss: 1.259114
(Iteration 191 / 200) loss: 1.409465
(Epoch 5 / 5) train acc: 0.548000; val_acc: 0.320647
```



13 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 `ece697ls/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.

[30]:
```
# Test RMSProp implementation
from ece662.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
```

[31]:
```
# Test Adam implementation
from ece662.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
```

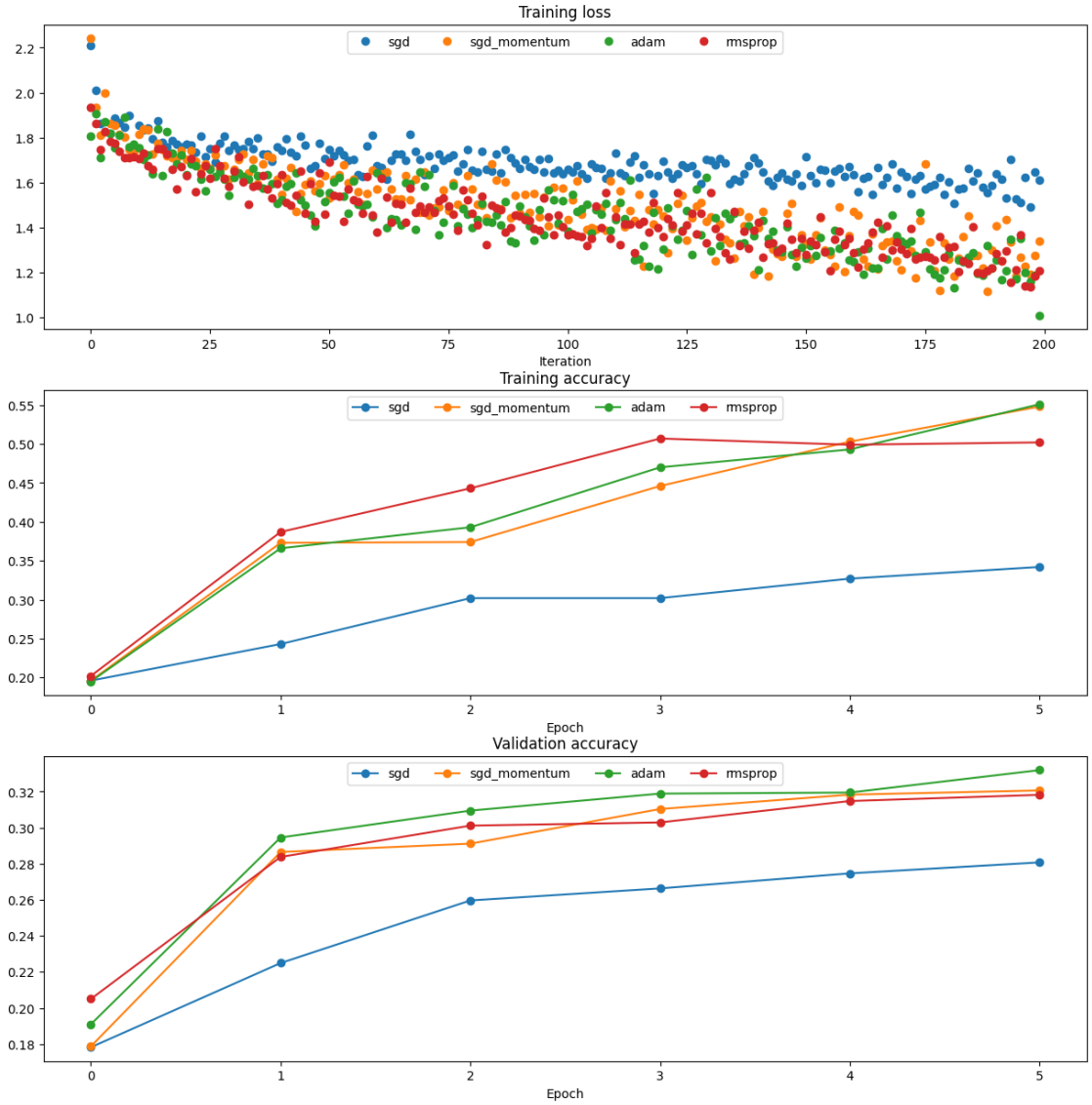
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[32]:
```
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,␣
 ↪num_classes=6)

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

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

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

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

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

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

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

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

```
running with  adam
(Iteration 1 / 200) loss: 1.807499
(Epoch 0 / 5) train acc: 0.195000; val_acc: 0.190976
(Iteration 11 / 200) loss: 1.753515
(Iteration 21 / 200) loss: 1.633546
(Iteration 31 / 200) loss: 1.622290
(Epoch 1 / 5) train acc: 0.366000; val_acc: 0.294556
(Iteration 41 / 200) loss: 1.522243
(Iteration 51 / 200) loss: 1.557197
(Iteration 61 / 200) loss: 1.641452
(Iteration 71 / 200) loss: 1.584083
(Epoch 2 / 5) train acc: 0.393000; val_acc: 0.309465
(Iteration 81 / 200) loss: 1.401046
(Iteration 91 / 200) loss: 1.495734
(Iteration 101 / 200) loss: 1.381813
(Iteration 111 / 200) loss: 1.434414
(Epoch 3 / 5) train acc: 0.470000; val_acc: 0.318882
(Iteration 121 / 200) loss: 1.302000
(Iteration 131 / 200) loss: 1.294296
```

```
(Iteration 141 / 200) loss: 1.213057
(Iteration 151 / 200) loss: 1.262926
(Epoch 4 / 5) train acc: 0.493000; val_acc: 0.319470
(Iteration 161 / 200) loss: 1.247106
(Iteration 171 / 200) loss: 1.340207
(Iteration 181 / 200) loss: 1.284272
(Iteration 191 / 200) loss: 1.254844
(Epoch 5 / 5) train acc: 0.551000; val_acc: 0.331927

running with  rmsprop
(Iteration 1 / 200) loss: 1.932862
(Epoch 0 / 5) train acc: 0.202000; val_acc: 0.205002
(Iteration 11 / 200) loss: 1.708115
(Iteration 21 / 200) loss: 1.629722
(Iteration 31 / 200) loss: 1.650252
(Epoch 1 / 5) train acc: 0.387000; val_acc: 0.283767
(Iteration 41 / 200) loss: 1.634964
(Iteration 51 / 200) loss: 1.690334
(Iteration 61 / 200) loss: 1.379711
(Iteration 71 / 200) loss: 1.467437
(Epoch 2 / 5) train acc: 0.443000; val_acc: 0.301128
(Iteration 81 / 200) loss: 1.462081
(Iteration 91 / 200) loss: 1.453007
(Iteration 101 / 200) loss: 1.368016
(Iteration 111 / 200) loss: 1.445894
(Epoch 3 / 5) train acc: 0.507000; val_acc: 0.302894
(Iteration 121 / 200) loss: 1.360662
(Iteration 131 / 200) loss: 1.554890
(Iteration 141 / 200) loss: 1.424701
(Iteration 151 / 200) loss: 1.279935
(Epoch 4 / 5) train acc: 0.499000; val_acc: 0.314762
(Iteration 161 / 200) loss: 1.288671
(Iteration 171 / 200) loss: 1.303293
(Iteration 181 / 200) loss: 1.312834
(Iteration 191 / 200) loss: 1.284701
(Epoch 5 / 5) train acc: 0.502000; val_acc: 0.318195
```



13.1 Inline Question 4:

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?

13.2 Answer:

We can see here the cache get increment by dw^2 everytime but w only update by sqrt(cache), thus make the learning speed really slow. Adam avoids this problem because instead of letting the cache grow without bound, it uses exponential moving averages of past gradients. That means it "forgets" older gradients over time, so the effective learning rate doesn't just keep shrinking forever.

14 Train a good model!

Train the best fully-connected model that you can on CINIC-10, storing your best model in the `best_model` variable. We require you to get at least 40% accuracy on the validation set using a fully-connected net.

Later in the assignment we will ask you to train the best convolutional network that you can on CINIC-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

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.

[8]:
```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CINIC-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)*****

best_model = FullyConnectedNet(
    hidden_dims=[1024, 512, 256, 128],
    input_dim=3 * 32 * 32,
    num_classes=6,
    dropout=0.4,
    normalization='batchnorm',
    reg=1e-3,
    weight_scale=1e-2,
    dtype=np.float32
)

solver = Solver(
    best_model,
    data,
    update_rule='adam',
    optim_config={
        'learning_rate': 3e-3,
    },
    lr_decay=0.9,
    batch_size=64,
    num_epochs=10,
```

```
    print_every=100,
    verbose=True
)
solver.train()

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

```
(Iteration 1 / 8430) loss: 1.995810
(Epoch 0 / 10) train acc: 0.178000; val_acc: 0.159392
(Iteration 101 / 8430) loss: 2.985755
(Iteration 201 / 8430) loss: 2.483694
(Iteration 301 / 8430) loss: 3.254369
(Iteration 401 / 8430) loss: 5.887314
(Iteration 501 / 8430) loss: 13.278814
(Iteration 601 / 8430) loss: 15.588056
(Iteration 701 / 8430) loss: 15.048997
(Iteration 801 / 8430) loss: 16.613806
(Epoch 1 / 10) train acc: 0.171000; val_acc: 0.167631
(Iteration 901 / 8430) loss: 20.240837
(Iteration 1001 / 8430) loss: 19.115175
(Iteration 1101 / 8430) loss: 18.615070
(Iteration 1201 / 8430) loss: 17.784853
(Iteration 1301 / 8430) loss: 17.038181
(Iteration 1401 / 8430) loss: 16.927412
(Iteration 1501 / 8430) loss: 16.314877
(Iteration 1601 / 8430) loss: 15.363873
(Epoch 2 / 10) train acc: 0.166000; val_acc: 0.180284
(Iteration 1701 / 8430) loss: 15.319561
(Iteration 1801 / 8430) loss: 14.796522
(Iteration 1901 / 8430) loss: 14.948975
(Iteration 2001 / 8430) loss: 14.878683
(Iteration 2101 / 8430) loss: 15.789031
(Iteration 2201 / 8430) loss: 15.513789
(Iteration 2301 / 8430) loss: 14.858605
(Iteration 2401 / 8430) loss: 15.237719
(Iteration 2501 / 8430) loss: 15.316893
(Epoch 3 / 10) train acc: 0.182000; val_acc: 0.201275
(Iteration 2601 / 8430) loss: 18.692732
(Iteration 2701 / 8430) loss: 17.733372
(Iteration 2801 / 8430) loss: 17.607990
(Iteration 2901 / 8430) loss: 17.997374
(Iteration 3001 / 8430) loss: 17.162165
(Iteration 3101 / 8430) loss: 16.450281
(Iteration 3201 / 8430) loss: 15.263810
(Iteration 3301 / 8430) loss: 15.640224
```

```
(Epoch 4 / 10) train acc: 0.194000; val_acc: 0.190584
(Iteration 3401 / 8430) loss: 15.298072
(Iteration 3501 / 8430) loss: 14.671037
(Iteration 3601 / 8430) loss: 14.821360
(Iteration 3701 / 8430) loss: 14.603885
(Iteration 3801 / 8430) loss: 14.447033
(Iteration 3901 / 8430) loss: 13.916535
(Iteration 4001 / 8430) loss: 13.842980
(Iteration 4101 / 8430) loss: 13.953815
(Iteration 4201 / 8430) loss: 14.669315
(Epoch 5 / 10) train acc: 0.161000; val_acc: 0.189308
(Iteration 4301 / 8430) loss: 14.169937
(Iteration 4401 / 8430) loss: 15.044115
(Iteration 4501 / 8430) loss: 15.596191
(Iteration 4601 / 8430) loss: 16.437801
(Iteration 4701 / 8430) loss: 15.683645
(Iteration 4801 / 8430) loss: 16.509211
(Iteration 4901 / 8430) loss: 19.224592
(Iteration 5001 / 8430) loss: 18.194471
(Epoch 6 / 10) train acc: 0.192000; val_acc: 0.192153
(Iteration 5101 / 8430) loss: 17.751310
(Iteration 5201 / 8430) loss: 20.071533
(Iteration 5301 / 8430) loss: 20.138000
(Iteration 5401 / 8430) loss: 19.469496
(Iteration 5501 / 8430) loss: 18.530523
(Iteration 5601 / 8430) loss: 19.881609
(Iteration 5701 / 8430) loss: 20.227060
(Iteration 5801 / 8430) loss: 19.467674
(Iteration 5901 / 8430) loss: 20.735239
(Epoch 7 / 10) train acc: 0.193000; val_acc: 0.201177
(Iteration 6001 / 8430) loss: 23.570829
(Iteration 6101 / 8430) loss: 21.781584
(Iteration 6201 / 8430) loss: 23.823666
(Iteration 6301 / 8430) loss: 23.706837
(Iteration 6401 / 8430) loss: 24.119967
(Iteration 6501 / 8430) loss: 24.242558
(Iteration 6601 / 8430) loss: 25.873692
(Iteration 6701 / 8430) loss: 23.569225
(Epoch 8 / 10) train acc: 0.205000; val_acc: 0.195292
(Iteration 6801 / 8430) loss: 24.133898
(Iteration 6901 / 8430) loss: 25.208073
(Iteration 7001 / 8430) loss: 25.816595
(Iteration 7101 / 8430) loss: 27.794781
(Iteration 7201 / 8430) loss: 27.348969
(Iteration 7301 / 8430) loss: 27.296064
(Iteration 7401 / 8430) loss: 26.444490
(Iteration 7501 / 8430) loss: 29.849899
(Epoch 9 / 10) train acc: 0.193000; val_acc: 0.202550
```

```
(Iteration 7601 / 8430) loss: 30.536005
(Iteration 7701 / 8430) loss: 28.855661
(Iteration 7801 / 8430) loss: 29.556919
(Iteration 7901 / 8430) loss: 31.980320
(Iteration 8001 / 8430) loss: 32.447571
(Iteration 8101 / 8430) loss: 32.042576
(Iteration 8201 / 8430) loss: 29.484144
(Iteration 8301 / 8430) loss: 29.245913
(Iteration 8401 / 8430) loss: 29.571671
(Epoch 10 / 10) train acc: 0.213000; val_acc: 0.203139
```

15 Test your model!

Run your best model on the validation and test sets. You should achieve above 40% accuracy on the validation set.

[6]:
```
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:  0.1795978420794507
Test set accuracy:  0.1889956845821891
```

Q2-BatchNorm

October 5, 2025

[4]:
```
# # this mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive', force_remount=True)

# # enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'ece697ls/assignments/assignment3/'
# 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))

# %cd /content
```

1 Batch Normalization

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

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

The authors of [1] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [1] proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated

means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

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

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

[5]:
```
# As usual, a bit of setup
import time
import numpy as np
import matplotlib.pyplot as plt
from ece662.classifiers.fc_net import *
from ece662.data_utils import get_CINIC10_data
from ece662.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from ece662.solver import Solver

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

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

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

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

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

[6]:
```
# Load the (preprocessed) CINIC10 data - Note that CINIC10 was modified in size␣
 ↪for this course

data = get_CINIC10_data()
```

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

print('Number of Classes: {}'.format(len(np.unique(data['y_train']))))
```

```
('X_train: ', (53973, 3, 32, 32))
('y_train: ', (53973,))
('X_val: ', (10195, 3, 32, 32))
('y_val: ', (10195,))
('X_test: ', (10196, 3, 32, 32))
('y_test: ', (10196,))
Number of Classes: 6
```

1.1 Batch normalization: forward

In the file `ece662ls/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!

[7]:
```
# 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:  [8.88178420e-18 2.27595720e-17 5.30825384e-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]
```

[8]:
```
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.

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

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

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

bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

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

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

1.2 Batch normalization: backward

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

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

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

[9]:
```
# 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:  1.0
dgamma error:  5.418458160170129e-12
dbeta error:  2.276445013433725e-12
```

1.3 Batch normalization: alternative backward

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

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

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

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

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

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

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

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

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

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

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

[10]:
```
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:  1.0
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.49x
```

1.4 Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `ece697ls/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 `ece697ls/layer_utils.py`. If you decide to do so, do it in the file `ece697ls/classifiers/fc_net.py`.

[11]:
```
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: 1.00e+00
W2 relative error: 9.99e-01
W3 relative error: 3.75e-10
b1 relative error: 3.55e-07
b2 relative error: 2.22e-08
b3 relative error: 9.06e-11
beta1 relative error: 1.00e+00
beta2 relative error: 2.67e-09
gamma1 relative error: 1.00e+00
gamma2 relative error: 2.41e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.00e+00
W2 relative error: 1.00e+00
W3 relative error: 1.11e-08
b1 relative error: 1.78e-07
b2 relative error: 4.44e-08
b3 relative error: 2.23e-10
beta1 relative error: 1.00e+00
beta2 relative error: 5.69e-09
gamma1 relative error: 1.00e+00
gamma2 relative error: 4.14e-09
```

2 Batchnorm for deep networks

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

[12]:
```
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', num_classes=6)
```

```
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
  ↪normalization=None, num_classes=6)

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()
```

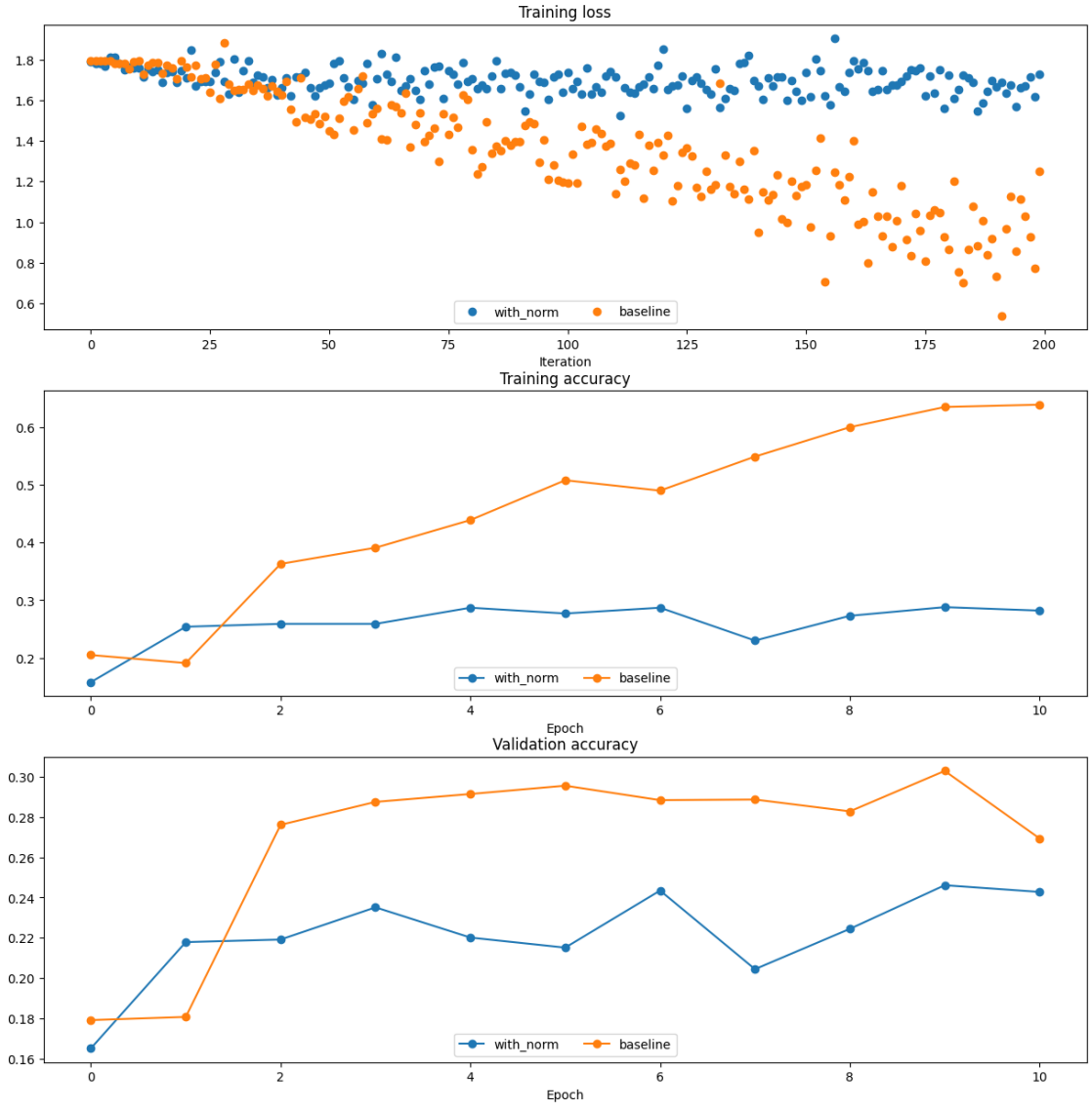
```
Solver with batch norm:
(Iteration 1 / 200) loss: 1.791199
(Epoch 0 / 10) train acc: 0.158000; val_acc: 0.165081
(Epoch 1 / 10) train acc: 0.254000; val_acc: 0.217852
(Iteration 21 / 200) loss: 1.710887
(Epoch 2 / 10) train acc: 0.259000; val_acc: 0.219127
(Iteration 41 / 200) loss: 1.663039
(Epoch 3 / 10) train acc: 0.259000; val_acc: 0.235115
(Iteration 61 / 200) loss: 1.703033
(Epoch 4 / 10) train acc: 0.287000; val_acc: 0.220108
(Iteration 81 / 200) loss: 1.704903
(Epoch 5 / 10) train acc: 0.277000; val_acc: 0.215105
(Iteration 101 / 200) loss: 1.734312
(Epoch 6 / 10) train acc: 0.287000; val_acc: 0.243453
(Iteration 121 / 200) loss: 1.850489
(Epoch 7 / 10) train acc: 0.230000; val_acc: 0.204414
(Iteration 141 / 200) loss: 1.670057
(Epoch 8 / 10) train acc: 0.273000; val_acc: 0.224424
(Iteration 161 / 200) loss: 1.795607
(Epoch 9 / 10) train acc: 0.288000; val_acc: 0.246101
(Iteration 181 / 200) loss: 1.720966
(Epoch 10 / 10) train acc: 0.282000; val_acc: 0.242766

Solver without batch norm:
(Iteration 1 / 200) loss: 1.791549
```

```
(Epoch 0 / 10) train acc: 0.205000; val_acc: 0.179107
(Epoch 1 / 10) train acc: 0.191000; val_acc: 0.180677
(Iteration 21 / 200) loss: 1.762525
(Epoch 2 / 10) train acc: 0.363000; val_acc: 0.276116
(Iteration 41 / 200) loss: 1.627670
(Epoch 3 / 10) train acc: 0.391000; val_acc: 0.287494
(Iteration 61 / 200) loss: 1.560715
(Epoch 4 / 10) train acc: 0.439000; val_acc: 0.291417
(Iteration 81 / 200) loss: 1.356394
(Epoch 5 / 10) train acc: 0.508000; val_acc: 0.295537
(Iteration 101 / 200) loss: 1.193580
(Epoch 6 / 10) train acc: 0.490000; val_acc: 0.288377
(Iteration 121 / 200) loss: 1.327613
(Epoch 7 / 10) train acc: 0.549000; val_acc: 0.288671
(Iteration 141 / 200) loss: 0.951490
(Epoch 8 / 10) train acc: 0.600000; val_acc: 0.282786
(Iteration 161 / 200) loss: 1.400837
(Epoch 9 / 10) train acc: 0.635000; val_acc: 0.302894
(Iteration 181 / 200) loss: 0.865792
(Epoch 10 / 10) train acc: 0.639000; val_acc: 0.269348
```

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.

[13]:
```
def plot_training_history(title, label, baseline, bn_solvers, plot_fn,␣
 ↪bl_marker='.', bn_marker='.', labels=None):
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
    bl_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


plt.subplot(3, 1, 1)
plot_training_history('Training loss','Iteration', solver, [bn_solver], \
                      lambda x: x.loss_history, bl_marker='o', bn_marker='o')
plt.subplot(3, 1, 2)
```

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

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



3 Batch normalization and initialization

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

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

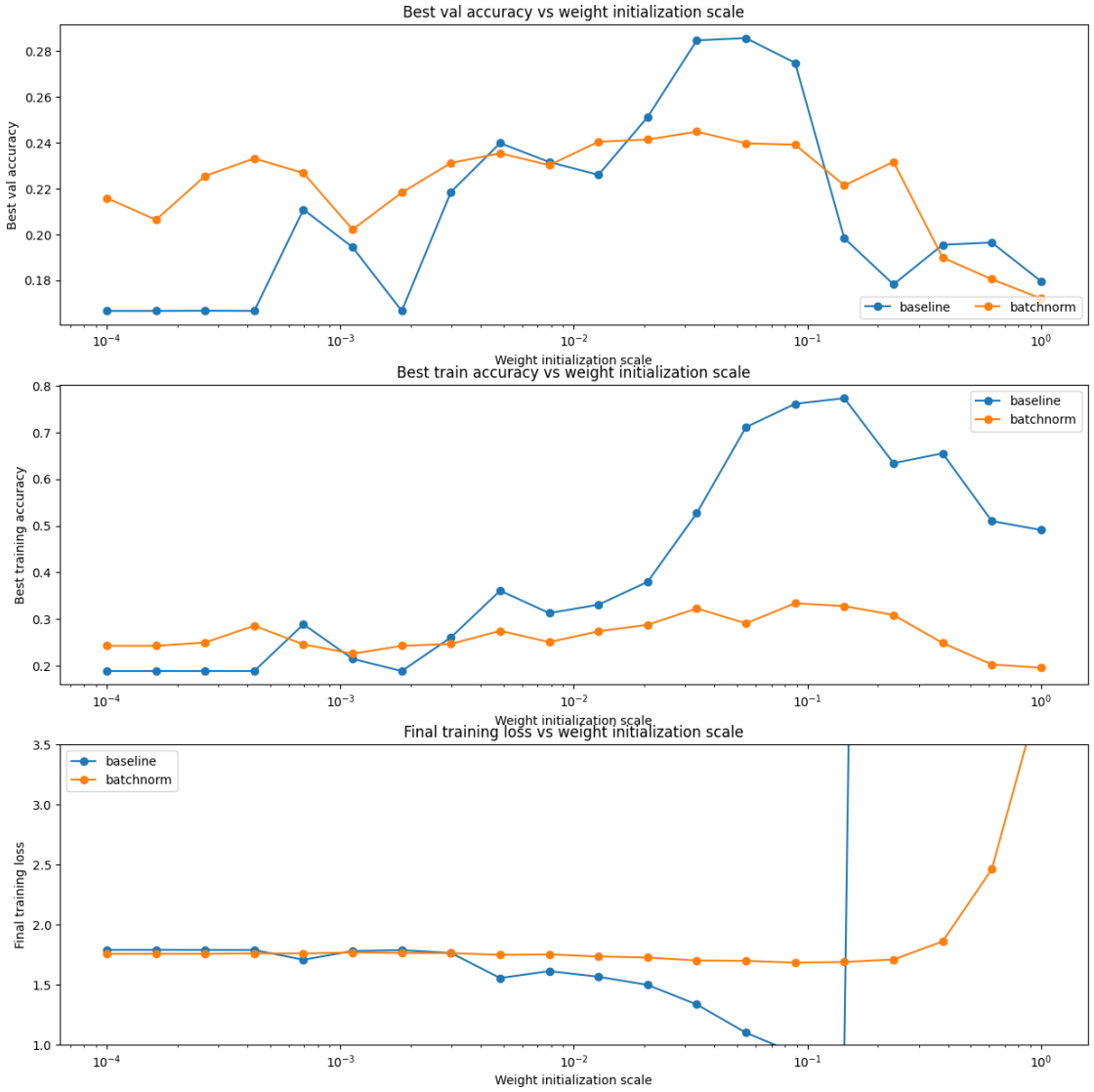
[14]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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', num_classes=6)
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None, num_classes=6)

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

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

```
    solvers_ws[weight_scale] = solver
```

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

[15]:
```
# Plot results of weight scale experiment
best_train_accs, bn_best_train_accs = [], []
best_val_accs, bn_best_val_accs = [], []
final_train_loss, bn_final_train_loss = [], []

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

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

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

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

plt.subplot(3, 1, 2)
```

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

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

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



3.1 Inline Question 1:

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

3.2 Answer:

The experiment shows that batch normalization makes the network much more robust to different weight initialization scales compared to networks without batch normalization.

Key observations: 1. **Without batch normalization**: The network is very sensitive to weight initialization scale. Performance drops significantly for both very small and very large weight scales, with optimal performance only in a narrow range around 0.01-0.1.

2. **With batch normalization**: The network maintains good performance across a much wider range of weight initialization scales, from very small (1e-4) to relatively large (1e-1) scales.

Why this happens: - **Without batch norm**: The network suffers from vanishing/exploding gradients at extreme weight scales. Small weights lead to vanishing gradients, while large weights cause exploding gradients and unstable training.

- **With batch norm**: The normalization step (subtracting mean, dividing by std) helps stabilize the activations regardless of the initial weight scale. This prevents both vanishing and exploding gradient problems, making the network much more robust to initialization.

The normalization effectively "rescales" the activations to have zero mean and unit variance, which acts as a form of implicit weight scaling that compensates for poor initialization choices.

4 Batch normalization and batch size

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

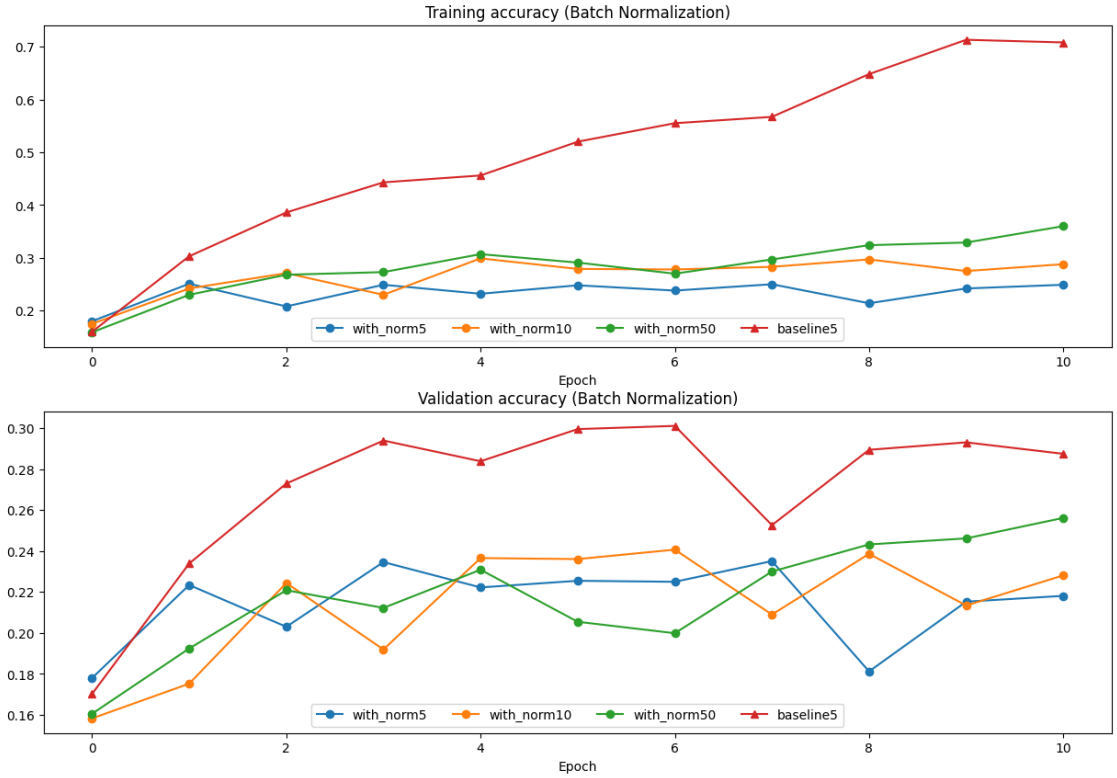
The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

[16]:
```
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, num_classes=6)
    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, num_classes=6)
        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
```

[17]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)

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



4.1 Inline Question 2:

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

4.2 Answer:

The experiment shows that batch normalization performance is highly dependent on batch size, with smaller batch sizes leading to worse performance.

Key observations: 1. **Small batch size (5)**: Batch normalization performs poorly, with low training and validation accuracy that doesn't improve much over time.

2. **Medium batch size (10)**: Performance improves but is still suboptimal.
3. **Larger batch size (50)**: Batch normalization works well, achieving good training and validation accuracy.
4. **No normalization**: Performance is relatively consistent across different batch sizes, though generally lower than batch norm with appropriate batch size.

Why this relationship exists: - **Statistical reliability**: Batch normalization computes mean and variance from the current batch. With small batches, these statistics are noisy and unreliable estimates of the true population statistics.

- **Normalization quality**: Small batches lead to poor normalization because the computed mean and variance don't represent the true distribution of the data, causing the normalization to be ineffective or even harmful.
- **Gradient quality**: The noisy statistics in small batches lead to noisy gradients, making training unstable and preventing the network from learning effectively.
- **Generalization**: The running averages used at test time are based on training-time statistics. If these are computed from noisy small-batch statistics, the test-time normalization will also be poor.

This is a fundamental limitation of batch normalization - it requires sufficiently large batches to compute reliable normalization statistics.

5 Layer Normalization

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

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

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

5.1 Inline Question 3:

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

1. Scaling each image in the dataset, so that the RGB channels for each row of pixels within an image sums up to 1.
2. Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1.
3. Subtracting the mean image of the dataset from each image in the dataset.
4. Setting all RGB values to either 0 or 1 depending on a given threshold.

5.2 Answer:

Batch Normalization: Option 3 - "Subtracting the mean image of the dataset from each image in the dataset."

Layer Normalization: Option 2 - "Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1."

Explanation: - **Batch normalization** normalizes across the batch dimension (across different samples). Option 3 subtracts the mean image computed from the entire dataset, which normalizes across all images in the dataset (batch dimension).

- **Layer normalization** normalizes across the feature dimension (within each sample). Option 2 normalizes each individual image so that all pixels within that image sum to 1, which is normalizing across the feature dimension (pixels) within each sample.
- Option 1 normalizes each row within an image, which is more like normalizing across a subset of features, not the full feature dimension.
- Option 4 is just thresholding/binarization, not normalization.

6 Layer Normalization: Implementation

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

Here's what you need to do:

- In `ece697ls/layers.py`, implement the forward pass for layer normalization in the function `layernorm_forward`.

Run the cell below to check your results. * In `ece697ls/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `ece697ls/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.

[18]:
```
# 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 -1.48029737e-16 -2.59052039e-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]
```

[19]:
```
# 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:  1.0
```

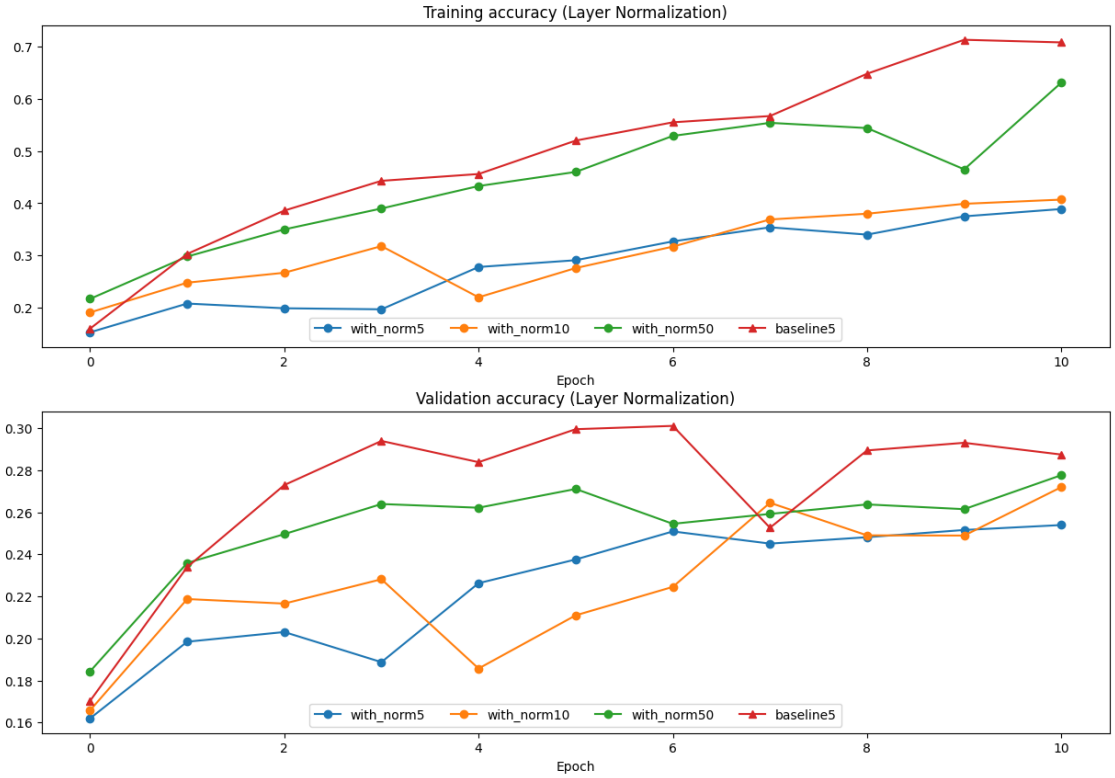
```
dgamma error:  1.9793843388564062e-12
dbeta error:  2.276445013433725e-12
```

7 Layer Normalization and batch size

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

[20]:
```
ln_solvers_bsize, solver_bsize, batch_sizes = ␣
 ↪run_batchsize_experiments('layernorm')

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

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

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



7.1 Inline Question 4:

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

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

7.2 Answer:

Option 2: Having a very small dimension of features

Why layer normalization doesn't work well with small feature dimensions:

1. **Statistical instability**: Layer normalization computes mean and variance across the feature dimension. With very few features (e.g., 2-3 features), these statistics become extremely noisy and unreliable.

2. **Poor normalization quality**: The mean and variance computed from just a few features don't provide meaningful normalization. The normalized values can become very large or very small, leading to unstable gradients.

3. **Loss of representational power**: With few features, the normalization step can over-constrain the network by forcing all features to have similar scales, reducing the network's ability to learn diverse representations.

4. **Gradient issues**: The noisy statistics lead to noisy gradients, making training unstable and preventing effective learning.

Why the other options are less problematic: - **Very deep networks**: Layer normalization actually works well in deep networks and is often preferred over batch normalization for this reason. - **High regularization**: While high regularization can hurt performance, it doesn't specifically interfere with layer normalization's mechanism.

Q3-Dropout

October 5, 2025

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

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'ece697ls/assignments/assignment3/'
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))

%cd /content
```

1 Dropout

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

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

[1]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from ece662.classifiers.fc_net import *
from ece662.data_utils import get_CINIC10_data
from ece662.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from ece662.solver import Solver

%matplotlib inline
```

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

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

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

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

[2]:
```
# Load the (preprocessed) CINIC10 data - Note that CINIC10 was modified in size␣
  ↪for this course

data = get_CINIC10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))

print('Number of Classes: {}'.format(len(np.unique(data['y_train']))))
```

```
('X_train: ', (53973, 3, 32, 32))
('y_train: ', (53973,))
('X_val: ', (10195, 3, 32, 32))
('y_val: ', (10195,))
('X_test: ', (10196, 3, 32, 32))
('y_test: ', (10196,))
Number of Classes: 6
```

2 Dropout forward pass

In the file `ece697ls/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

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

[3]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

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

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

```
Running tests with p =  0.25
Mean of input:  10.0002078784775
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.0002078784775
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:  10.0002078784775
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.0002078784775
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:  10.0002078784775
Mean of train-time output:  9.987811912159428
Mean of test-time output:  10.0002078784775
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 `ece697ls/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

[4]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
```

```
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,␣
 ↪dropout_param)[0], x, dout)

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

```
dx relative error:  1.8928938043362133e-11
```

3.1 Inline Question 1:

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

3.2 Answer:

If we don't divide the values by `p`, this makes training activations smaller than inference activations. Dividing by `p` fixes this by preserving the expected activation scale across training and inference.

4 Fully-connected nets with Dropout

In the file `ece697ls/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the network receives a value that is not 1 for the `dropout` parameter, then the net should add a dropout layer immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

[5]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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

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

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

```
  print()
```

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

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

Running check with dropout =  0.5
Initial loss:  2.3042759220785896
W1 relative error: 3.11e-07
W2 relative error: 1.84e-08
W3 relative error: 5.35e-08
b1 relative error: 2.58e-08
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

5 Regularization experiment

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

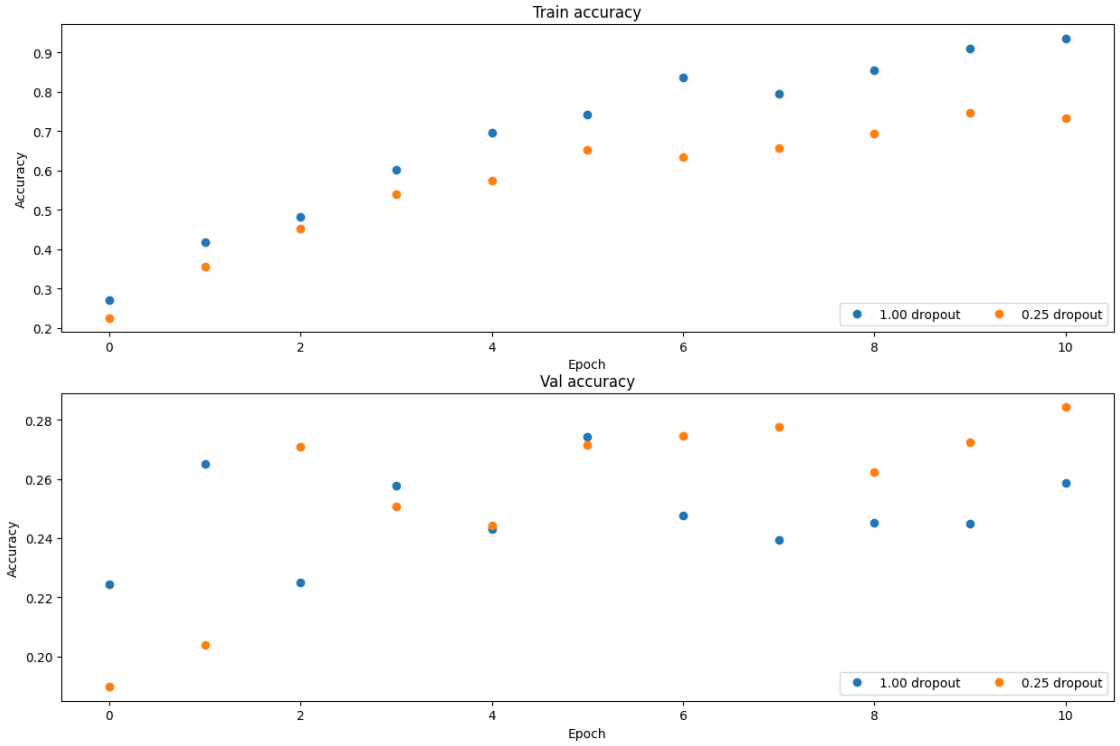
[6]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
```

```
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout, num_classes=6)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=10, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
  print()
```

```
1
(Iteration 1 / 50) loss: 6.164760
(Epoch 0 / 10) train acc: 0.270000; val_acc: 0.224326
(Epoch 1 / 10) train acc: 0.418000; val_acc: 0.265130
(Epoch 2 / 10) train acc: 0.482000; val_acc: 0.224914
(Epoch 3 / 10) train acc: 0.602000; val_acc: 0.257675
(Epoch 4 / 10) train acc: 0.696000; val_acc: 0.243060
(Epoch 5 / 10) train acc: 0.742000; val_acc: 0.274350
(Epoch 6 / 10) train acc: 0.836000; val_acc: 0.247572
(Epoch 7 / 10) train acc: 0.796000; val_acc: 0.239529
(Epoch 8 / 10) train acc: 0.856000; val_acc: 0.245120
(Epoch 9 / 10) train acc: 0.910000; val_acc: 0.245022
(Epoch 10 / 10) train acc: 0.936000; val_acc: 0.258754

0.25
(Iteration 1 / 50) loss: 10.595271
(Epoch 0 / 10) train acc: 0.226000; val_acc: 0.189799
(Epoch 1 / 10) train acc: 0.356000; val_acc: 0.204022
(Epoch 2 / 10) train acc: 0.452000; val_acc: 0.270721
(Epoch 3 / 10) train acc: 0.540000; val_acc: 0.250809
(Epoch 4 / 10) train acc: 0.574000; val_acc: 0.244139
(Epoch 5 / 10) train acc: 0.652000; val_acc: 0.271506
(Epoch 6 / 10) train acc: 0.634000; val_acc: 0.274644
(Epoch 7 / 10) train acc: 0.658000; val_acc: 0.277685
(Epoch 8 / 10) train acc: 0.694000; val_acc: 0.262187
(Epoch 9 / 10) train acc: 0.746000; val_acc: 0.272388
(Epoch 10 / 10) train acc: 0.734000; val_acc: 0.284159
```

[7]:
```
# Plot train and validation accuracies of the two models
```

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

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

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

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



5.1 Inline Question 2:

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

5.2 Answer:

For training accuracy, adding dropout will definitely lower the accuracy as it will make the problems harder, adding more variable to the optimization problems(randomly drop units), preventing the network from memorizing the datasets. Thus for training accuracy, the one with dropout will have a lower accuracy. However, as it will generalize better, we can see that for validation accuracy, the one with dropout generalize better and perform better than the one without dropout. Dropout as a regularizer will sacrifice training accuracy in exchange for generalization.

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:

I think we should keep `p` the same when decided to decrease the size of hidden layers. The idea of dropout is randomly dropping nodes to avoid overfitting, however, we already decided ot decrease

the size of hidden layers, which means reducing the network capacity. Thus lowering `p` will make the network underfitting as we reduce too much network capacity at the same time. To balance out, we should keep the `p` the same or even increase it a little to make up for all the reduced capacity by decreasing hidden layers.

Q4-CNNs

October 5, 2025

[52]:
```
# # this mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive', force_remount=True)

# # enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'ece697ls/assignments/assignment3/'
# 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))

# %cd /content
```

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 CINIC-10 dataset.

[53]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from ece662.classifiers.cnn import *
from ece662.data_utils import get_CINIC10_data
from ece662.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from ece662.layers import *
from ece662.fast_layers import *
from ece662.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))))
```

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

[54]:
```
# Load the (preprocessed) CINIC10 data - Note that CINIC10 was modified in size␣
  ↪for this course

data = get_CINIC10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))

print('Number of Classes: {}'.format(len(np.unique(data['y_train']))))
```

```
('X_train: ', (53973, 3, 32, 32))
('y_train: ', (53973,))
('X_val: ', (10195, 3, 32, 32))
('y_val: ', (10195,))
('X_test: ', (10196, 3, 32, 32))
('y_test: ', (10196,))
Number of Classes: 6
```

2 Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `ece697ls/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:

[55]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
```

```
b = np.linspace(-0.1, 0.2, num=3)

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

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

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

3 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.

3.1 Colab Users Only

Please execute the below cell to copy two cat images to the Colab VM.

[56]:
```
# Colab users only!
%mkdir -p ece697ls/notebook_images
%cd drive/My\ Drive/$FOLDERNAME/ece697ls
%cp -r notebook_images/ /content/ece697ls/
%cd /content/
```

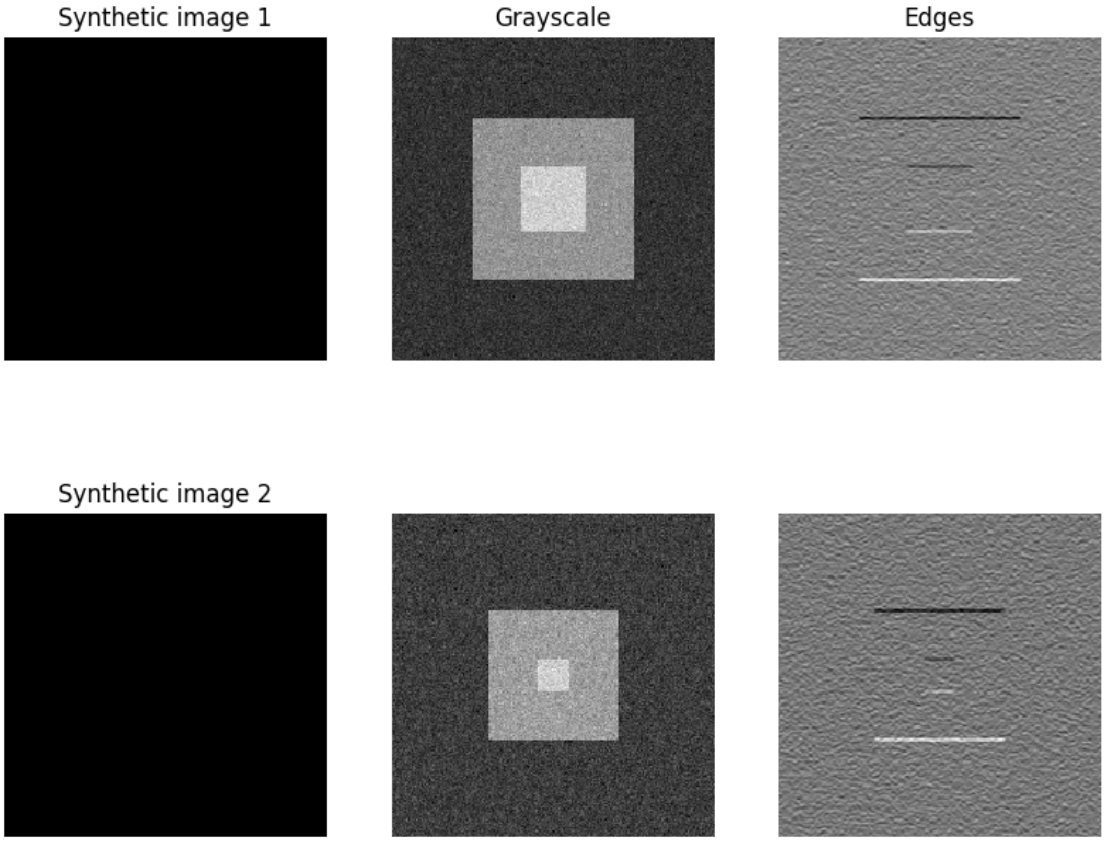
```
[Errno 2] No such file or directory: 'drive/My Drive/$FOLDERNAME/ece697ls'
/Users/tungngo/ece562-assignment1
cp: directory /content/ece697ls does not exist
[Errno 2] No such file or directory: '/content/'
/Users/tungngo/ece562-assignment1
```

[57]:
```
# Note: This cell requires image files that are not available in local␣
 ↪development
# For demonstration purposes, we'll create synthetic images instead

try:
    from imageio.v2 import imread  # Use v2 to avoid deprecation warning
    from PIL import Image

    # Try to load the images, but fall back to synthetic data if not available
    kitten = imread('ece697ls/notebook_images/kitten.jpg')
    puppy = imread('ece697ls/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))

except FileNotFoundError:
    print("Image files not found. Using synthetic data for demonstration.")
    # Create synthetic images for demonstration
    img_size = 200
    x = np.zeros((2, 3, img_size, img_size))

    # Create a synthetic "kitten" image with some patterns
    for i in range(3):
        x[0, i, :, :] = np.random.randn(img_size, img_size) * 0.1
        # Add some structure
        x[0, i, 50:150, 50:150] += 0.5
        x[0, i, 80:120, 80:120] += 0.3

    # Create a synthetic "puppy" image
    for i in range(3):
        x[1, i, :, :] = np.random.randn(img_size, img_size) * 0.1
        # Add some structure
        x[1, i, 60:140, 60:140] += 0.4
        x[1, i, 90:110, 90:110] += 0.2

# 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)
if 'puppy' in locals():
    imshow_no_ax(puppy, normalize=False)
    plt.title('Original image (puppy)')
else:
    imshow_no_ax(x[0].transpose(1, 2, 0), normalize=False)
    plt.title('Synthetic image 1')
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)
if 'kitten_cropped' in locals():
    imshow_no_ax(kitten_cropped, normalize=False)
    plt.title('Original image (kitten)')
else:
    imshow_no_ax(x[1].transpose(1, 2, 0), normalize=False)
    plt.title('Synthetic image 2')
```

```
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_no_ax(out[1, 1])
plt.show()
```

```
Image files not found. Using synthetic data for demonstration.
```



4 Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `ece697ls/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.

[58]:
```
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:  1.7808556723954787e-08
dw error:  3.2738772890979675e-10
db error:  4.473566414430221e-11
```

5 Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `ece697ls/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

[59]:
```
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
```

6 Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `ece697ls/layers.py`. You don't need to worry about computational efficiency.

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

[60]:
```
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
```

7 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 `ece697ls/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it either execute the local development cell (option A) if you are developing locally, or the Colab cell (option B) if you are running this assignment in Colab.

Very Important, Please Read. For **both** option A and B, you have to **restart** the notebook after compiling the cython extension. In Colab, please save the notebook `File -> Save`, then click

`Runtime -> Restart Runtime -> Yes`. This will restart the kernel which means local variables will be lost. Just re-execute the cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

7.1 Option A: Local Development

Go to the ece697ls directory and execute the following in your terminal:

```
python setup.py build_ext --inplace
```

7.2 Option B: Colab

Execute the cell below only only **ONCE**.

Note that you may need to terminate this session and start a new one after executing the code below

[61]:
```
%cd drive/My\ Drive/$FOLDERNAME/ece697ls/
!python setup.py build_ext --inplace
```

```
[Errno 2] No such file or directory: 'drive/My Drive/$FOLDERNAME/ece697ls/'
/Users/tungngo/ece562-assignment1
/opt/homebrew/Cellar/python@3.13/3.13.7/Frameworks/Python.framework/Versions/3.1
3/Resources/Python.app/Contents/MacOS/Python: can't open file
'/Users/tungngo/ece562-assignment1/setup.py': [Errno 2] No such file or
directory
```

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:

[62]:
```
# Rel errors should be around e-9 or less
from ece662.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
```

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

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

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

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

```
Testing conv_forward_fast:
Naive: 1.201577s
Fast: 0.002469s
Speedup: 486.653051x
Difference:  7.88225256277888e-11

Testing conv_backward_fast:
Naive: 1.673026s
Fast: 0.153910s
Speedup: 10.870146x
dx difference:  1.50193951894836e-11
dw difference:  2.538300430922586e-13
db difference:  0.0
```

[63]:
```
# Relative errors should be close to 0.0
from ece662.fast_layers import conv_forward_fast, conv_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: 0.088727s
fast: 0.001313s
speedup: 67.577084x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.253351s
fast: 0.003537s
speedup: 71.630064x
dx difference:  0.0
```

8 Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `ece697ls/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 they're working.

[64]:
```
from ece662.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:  1.3129103487372312e-08
dw error:  4.243260484602694e-09
db error:  3.579585075287619e-10
```

[65]:
```
from ece662.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:  5.532043677351017e-09
dw error:  2.556001426075349e-10
db error:  1.1891655515776076e-10
```

9 Three-layer ConvNet

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

Open the file `ece697ls/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:

9.1 Sanity check loss

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

[66]:
```
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):  2.302586071243987
Initial loss (with regularization):  2.5082556233317708
```

9.2 Gradient check

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

[67]:
```
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: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10
```

9.3 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.

[68]:
```
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, num_classes=6)

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: 1.833348
(Epoch 0 / 15) train acc: 0.390000; val_acc: 0.214321
(Iteration 2 / 30) loss: 2.642196
(Epoch 1 / 15) train acc: 0.440000; val_acc: 0.243256
(Iteration 3 / 30) loss: 1.948378
(Iteration 4 / 30) loss: 1.600015
(Epoch 2 / 15) train acc: 0.490000; val_acc: 0.226091
(Iteration 5 / 30) loss: 1.568312
(Iteration 6 / 30) loss: 1.371931
(Epoch 3 / 15) train acc: 0.580000; val_acc: 0.226287
(Iteration 7 / 30) loss: 1.112932
(Iteration 8 / 30) loss: 1.141561
(Epoch 4 / 15) train acc: 0.660000; val_acc: 0.241099
(Iteration 9 / 30) loss: 1.074416
(Iteration 10 / 30) loss: 1.123802
(Epoch 5 / 15) train acc: 0.730000; val_acc: 0.249240
(Iteration 11 / 30) loss: 0.742874
(Iteration 12 / 30) loss: 0.795579
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.248847
(Iteration 13 / 30) loss: 0.666708
(Iteration 14 / 30) loss: 0.914556
(Epoch 7 / 15) train acc: 0.870000; val_acc: 0.265915
(Iteration 15 / 30) loss: 0.609670
(Iteration 16 / 30) loss: 0.452952
(Epoch 8 / 15) train acc: 0.860000; val_acc: 0.253948
(Iteration 17 / 30) loss: 0.384983
(Iteration 18 / 30) loss: 0.497716
(Epoch 9 / 15) train acc: 0.860000; val_acc: 0.241491
(Iteration 19 / 30) loss: 0.416887
(Iteration 20 / 30) loss: 0.252471
(Epoch 10 / 15) train acc: 0.950000; val_acc: 0.262580
(Iteration 21 / 30) loss: 0.202691
(Iteration 22 / 30) loss: 0.110400
(Epoch 11 / 15) train acc: 0.910000; val_acc: 0.273664
(Iteration 23 / 30) loss: 0.245397
(Iteration 24 / 30) loss: 0.213571
(Epoch 12 / 15) train acc: 0.970000; val_acc: 0.275527
(Iteration 25 / 30) loss: 0.092706
(Iteration 26 / 30) loss: 0.075215
(Epoch 13 / 15) train acc: 0.950000; val_acc: 0.265424
(Iteration 27 / 30) loss: 0.110792
(Iteration 28 / 30) loss: 0.086366
(Epoch 14 / 15) train acc: 0.980000; val_acc: 0.264836
(Iteration 29 / 30) loss: 0.109698
```

```
(Iteration 30 / 30) loss: 0.038454
(Epoch 15 / 15) train acc: 0.970000; val_acc: 0.257381
```

[69]:
```python
# Print final training accuracy
print(
    "Small data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
```

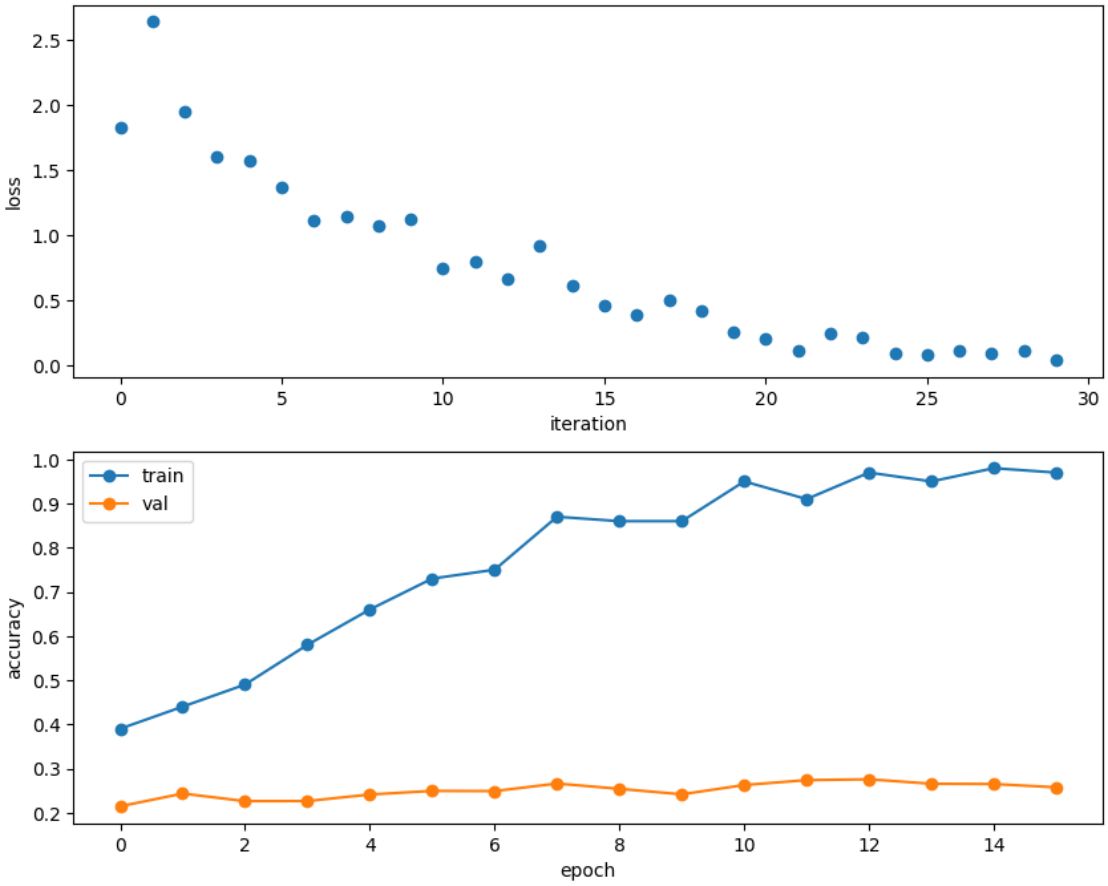
```
Small data training accuracy: 0.97
```

[70]:
```python
# Print final validation accuracy
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

```
Small data validation accuracy: 0.27552721922511036
```

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

[71]:
```python
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()
```



9.4 Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 35+% accuracy on the training set:

[72]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001,␣
  ↪num_classes=6)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=100)
solver.train()
```

```
(Iteration 1 / 1079) loss: 1.793714
(Epoch 0 / 1) train acc: 0.156000; val_acc: 0.166650
(Iteration 101 / 1079) loss: 1.697721
```

```
(Iteration 201 / 1079) loss: 1.628506
(Iteration 301 / 1079) loss: 1.789660
(Iteration 401 / 1079) loss: 1.601130
(Iteration 501 / 1079) loss: 1.408913
(Iteration 601 / 1079) loss: 1.609514
(Iteration 701 / 1079) loss: 1.522902
(Iteration 801 / 1079) loss: 1.463672
(Iteration 901 / 1079) loss: 1.486428
(Iteration 1001 / 1079) loss: 1.693357
(Epoch 1 / 1) train acc: 0.368000; val_acc: 0.371260
```

[73]:
```
# Print final training accuracy
print(
    "Full data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
```

```
Full data training accuracy: 0.35
```

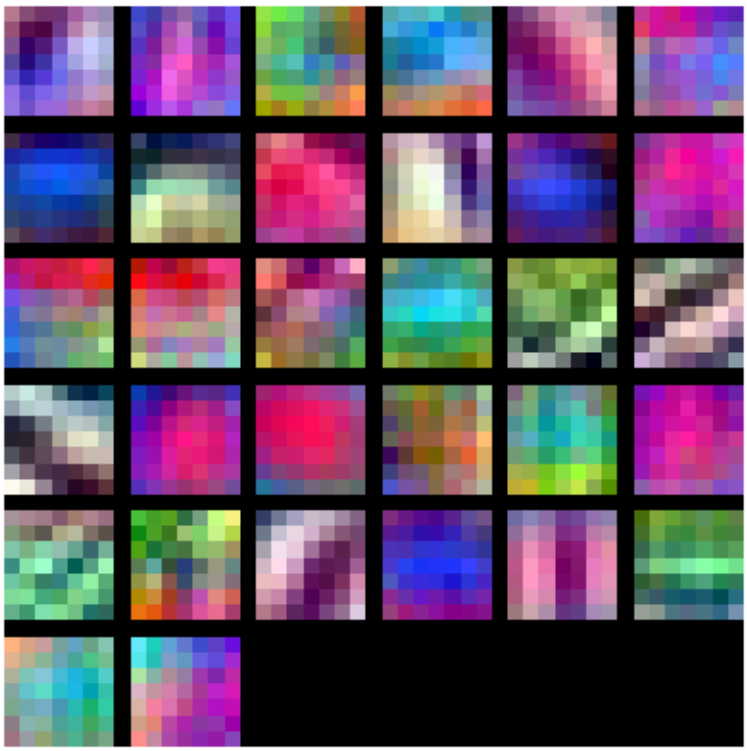
[74]:
```
# Print final validation accuracy
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
```

```
Full data validation accuracy: 0.3712604217753801
```

9.5 Visualize Filters

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

[76]:
```
from ece662.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()
```



10 Spatial Batch Normalization

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

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

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

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

10.1 Spatial batch normalization: forward

In the file `ece6971s/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

[77]:
```
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:  [ 5.96744876e-16  5.55111512e-16 -1.72084569e-16]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

[78]:
```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

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

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

10.2 Spatial batch normalization: backward

In the file `ece697ls/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:

[79]:
```
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:  1.0
dgamma error:  7.0969976381299756e-12
dbeta error:  3.2754947613321483e-12
```

11 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. Note that this normalization technique that you are to implement in the following cells was introduced and published to ECCV just in 2018 – this truly is still an ongoing and excitingly active field of research!

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

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

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

11.1 Group normalization: forward

In the file `ece697ls/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

[80]:
```
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:  [ 2.62752782e-16  4.66293670e-16  2.72929827e-16 -3.68223970e-16]
  Stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

11.2 Spatial group normalization: backward

In the file `ece697ls/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:

[81]:
```
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-12~1e-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:  6.616283351590772e-08
dgamma error:  1.0
dbeta error:  1.0
```

Q5-Pytorch

October 5, 2025

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

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'ece697ls/assignments/assignment3/'
FOLDERNAME = "Sandip/assignment1_colab/assignment1"
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))

%cd /content
```

1 What's this PyTorch business?

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

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

1.0.1 What is PyTorch?

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

1.0.2 Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).

- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.0.3 PyTorch versions

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

1.1 How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

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

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

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

2 Table of Contents

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

1. Part I, Preparation: we will a modified CINIC-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, CINIC-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CINIC-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

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

3 Part I. Preparation

First, we load the CINIC-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 CINIC-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

[2]:
```
import torch
#assert '.'.join(torch.__version__.split('.')[:2]) == '1.5'
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
from ece662.data_utils import get_CINIC10_data

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

import numpy as np
```

[3]:
```
NUM_TRAIN = 49000

data = get_CINIC10_data()

def prepare_dataloader(x,y):
    dset = []
    for i in range(len(y)):
        dset.append((x[i],y[i]))

    return dset

# 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.

loader_train = DataLoader(prepare_dataloader(data['X_train'],data['y_train']),␣
 ↪batch_size=64)
loader_val = DataLoader(prepare_dataloader(data['X_val'],data['y_val']),␣
 ↪batch_size=64)
```

```
loader_test = DataLoader(prepare_dataloader(data['X_test'],data['y_test']),␣
↪batch_size=64)
```

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

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

3.1 Colab Users

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

[6]:
```
USE_GPU = True

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

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

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

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

```
using device: cpu
```

4 Part 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 CINIC classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

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

4.0.1 PyTorch Tensors: Flatten Function

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

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

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

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

[9]:
```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector␣
 ↪per image

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

test_flatten()
```

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


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

4.0.2 Barebones PyTorch: Two-Layer Network

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

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

[10]:
```
import torch.nn.functional as F  # useful stateless functions

dtype = torch.float32

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, 6), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 6]

two_layer_fc_test()
```

```
torch.Size([64, 6])
```

4.0.3 Barebones PyTorch: Three-Layer ConvNet

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

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

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

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

[9]:
```
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)*****
  x = F.conv2d(x, conv_w1, bias=conv_b1)
  x = F.relu(x)
  x = F.conv2d(x, conv_w2, bias=conv_b2)
  x = F.relu(x)

  x = torch.flatten(x, start_dim=1)
  scores = x.mm(fc_w[:x.size(1), :]) + fc_b

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

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

When you run this function, scores should have shape (64, 6).

[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, 6))
    fc_b = torch.zeros(6)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w,␣
↪fc_b])
    print(scores.size())  # you should see [64, 6]
three_layer_convnet_test()
```

```
torch.Size([64, 6])
```

4.0.4 Barebones PyTorch: Initialization

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

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

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

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

[21]:
```
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, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, 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))
```

[21]:
```
tensor([[-0.2812, -1.4287, -1.6051, -0.1842,  0.5149],
        [-1.3148,  0.9984,  1.1611, -0.2777,  0.6200],
        [ 0.9384,  0.6007,  0.2401,  1.6701,  0.3927]], requires_grad=True)
```

4.0.5 Barebones PyTorch: Check Accuracy

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

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

[13]:
```
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(dtype=dtype)  # move to device, e.g. GPU
            y = y.to(dtype=torch.int64)
            scores = model_fn(x, params)
```

```
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *␣
↪acc))
```

4.0.6 BareBones PyTorch: Training Loop

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

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

[18]:
```
print_every = 100
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CINIC-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(dtype=dtype)
        y = y.to(dtype=torch.long)

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

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

        # Update parameters. We don't want to backpropagate through the
```

```
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

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

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

4.0.7 BareBones PyTorch: Train a Two-Layer Network

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

Each minibatch of CINIC 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 6-dimensional vector that represents the probability distribution over 6 classes.

You don't need to tune any hyperparameters but you should see accuracies above 30% after training for one epoch.

[19]:
```
hidden_layer_size = 4000
learning_rate = 1e-4

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 6))

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

```
Iteration 0, loss = 99.9762
Got 1921 / 10195 correct (18.84%)

Iteration 100, loss = 46.8981
Got 2564 / 10195 correct (25.15%)

Iteration 200, loss = 66.1544
Got 2917 / 10195 correct (28.61%)

Iteration 300, loss = 51.1496
Got 2598 / 10195 correct (25.48%)
```

```
Iteration 400, loss = 51.3450
Got 2796 / 10195 correct (27.43%)

Iteration 500, loss = 41.5020
Got 2906 / 10195 correct (28.50%)

Iteration 600, loss = 41.2783
Got 2975 / 10195 correct (29.18%)

Iteration 700, loss = 54.2181
Got 3491 / 10195 correct (34.24%)

Iteration 800, loss = 54.3007
Got 3086 / 10195 correct (30.27%)
```

4.0.8 BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CINIC. 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 6 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.

[22]:
```
learning_rate = 1e-4

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)*****
```

```
conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight(channel_1)

conv_w2 = random_weight((channel_2, channel_1, 3, 3))
conv_b2 = zero_weight(channel_2)

fc_w = random_weight((channel_2 * 32 * 32, 6))
fc_b = zero_weight(6)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                  END OF YOUR CODE                            #
################################################################################

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

```
Iteration 0, loss = 92.8009
Got 1710 / 10195 correct (16.77%)

Iteration 100, loss = 8.9809
Got 1999 / 10195 correct (19.61%)

Iteration 200, loss = 4.2008
Got 1927 / 10195 correct (18.90%)

Iteration 300, loss = 2.5483
Got 1904 / 10195 correct (18.68%)

Iteration 400, loss = 2.2312
Got 1888 / 10195 correct (18.52%)

Iteration 500, loss = 2.2228
Got 1876 / 10195 correct (18.40%)

Iteration 600, loss = 2.1172
Got 1849 / 10195 correct (18.14%)

Iteration 700, loss = 2.0389
Got 1803 / 10195 correct (17.69%)

Iteration 800, loss = 1.8974
Got 1807 / 10195 correct (17.72%)
```

5 Part III. PyTorch Module API

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

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

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

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

5.0.1 Module API: Two-Layer Network

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

[23]:
```
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, 6)
    scores = model(x)
    print(scores.size())  # you should see [64, 6]
test_TwoLayerFC()
```

```
torch.Size([64, 6])
```

5.0.2 Module API: Three-Layer ConvNet

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

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

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

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

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 6)` for the shape of the output scores.

[24]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
        # architecture defined above.                                          #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2)

        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1)

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

        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.kaiming_normal_(self.fc.weight)

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

```
        #                               END OF YOUR CODE                       ␣
 ↪#
        ########################################################################

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

        x = torch.relu(self.conv1(x))

        x = torch.relu(self.conv2(x))

        x = x.view(x.size(0), -1)  # or x.flatten(1)

        scores = self.fc(x)

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


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
 ↪size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8,␣
 ↪num_classes=6)
    scores = model(x)
    print(scores.size())  # you should see [64, 6]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 6])
```

5.0.3 Module API: Check Accuracy

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

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

[12]:
```
def check_accuracy_part34(loader, model):
```

```
    print('Checking accuracy')
    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(dtype=dtype)  # move to device, e.g. GPU
            y = y.to(dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 *␣
↪acc))
```

5.0.4 Module API: Training Loop

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

[4]:
```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CINIC-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()  # 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(dtype=dtype)  # move to device, e.g. GPU
            y = y.to(dtype=torch.long)

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

            # Zero out all of the gradients for the variables which the␣
↪optimizer
```

```
            # will update.
            optimizer.zero_grad()

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

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

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

5.0.5 Module API: Train a Two-Layer Network

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

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

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

You don't need to tune any hyperparameters, but you should see model accuracies above 30% after training for one epoch.

[32]:
```
hidden_layer_size = 4000
learning_rate = 1e-4
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 6)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 94.0595
Checking accuracy
Got 1765 / 10195 correct (17.31)

Iteration 100, loss = 62.8795
Checking accuracy
Got 2256 / 10195 correct (22.13)

Iteration 200, loss = 47.7527
Checking accuracy
Got 2711 / 10195 correct (26.59)

Iteration 300, loss = 53.6197
Checking accuracy
```

```
Got 2815 / 10195 correct (27.61)

Iteration 400, loss = 72.4728
Checking accuracy
Got 2491 / 10195 correct (24.43)

Iteration 500, loss = 62.3968
Checking accuracy
Got 2835 / 10195 correct (27.81)

Iteration 600, loss = 40.3097
Checking accuracy
Got 2779 / 10195 correct (27.26)

Iteration 700, loss = 56.3144
Checking accuracy
Got 3203 / 10195 correct (31.42)

Iteration 800, loss = 37.6415
Checking accuracy
Got 3004 / 10195 correct (29.47)
```

5.0.6 Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CINIC. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters.

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

[33]:
```
learning_rate = 3e-6
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = ThreeLayerConvNet(in_channel=3, channel_1=channel_1,␣
 ↪channel_2=channel_2, num_classes=6)

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

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

```
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 102.6038
Checking accuracy
Got 1520 / 10195 correct (14.91)

Iteration 100, loss = 39.5336
Checking accuracy
Got 1852 / 10195 correct (18.17)

Iteration 200, loss = 31.8100
Checking accuracy
Got 2040 / 10195 correct (20.01)

Iteration 300, loss = 34.0880
Checking accuracy
Got 2175 / 10195 correct (21.33)

Iteration 400, loss = 33.2105
Checking accuracy
Got 2242 / 10195 correct (21.99)

Iteration 500, loss = 32.6998
Checking accuracy
Got 2275 / 10195 correct (22.31)

Iteration 600, loss = 28.7388
Checking accuracy
Got 2382 / 10195 correct (23.36)

Iteration 700, loss = 27.8698
Checking accuracy
Got 2400 / 10195 correct (23.54)

Iteration 800, loss = 29.8988
Checking accuracy
Got 2462 / 10195 correct (24.15)
```

6 Part IV. PyTorch Sequential API

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

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

`forward()`. Is there a more convenient way?

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

6.0.1 Sequential API: Two-Layer Network

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

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 30% accuracy after one epoch of training.

[34]:
```
# 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-4

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 6),
)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 17.7208
Checking accuracy
Got 1777 / 10195 correct (17.43)

Iteration 100, loss = 17.1551
Checking accuracy
Got 2420 / 10195 correct (23.74)

Iteration 200, loss = 12.7181
Checking accuracy
Got 2774 / 10195 correct (27.21)

Iteration 300, loss = 15.9507
```

```
Checking accuracy
Got 2754 / 10195 correct (27.01)

Iteration 400, loss = 8.8443
Checking accuracy
Got 2998 / 10195 correct (29.41)

Iteration 500, loss = 12.4267
Checking accuracy
Got 2857 / 10195 correct (28.02)

Iteration 600, loss = 10.4857
Checking accuracy
Got 3105 / 10195 correct (30.46)

Iteration 700, loss = 11.0264
Checking accuracy
Got 3114 / 10195 correct (30.54)

Iteration 800, loss = 13.0916
Checking accuracy
Got 3060 / 10195 correct (30.01)
```

6.0.2 Sequential API: Three-Layer ConvNet

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

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 6 classes

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

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

Again, you don't need to tune any hyperparameters but you should see accuracy above 35% after one epoch of training.

[35]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-4

model = None
optimizer = None
```

```
################################################################################
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################
model = nn.Sequential( nn.Conv2d(3, channel_1, 5, padding = 2),
                       nn.ReLU(),
                       nn.Conv2d(channel_1, channel_2, 3, padding = 1),
                       nn.ReLU(),
                       Flatten(),
                       nn.Linear(channel_2 * 32 * 32, 6)
                     )
optimizer = optim.SGD(model.parameters(), lr = learning_rate, momentum = 0.9,␣
 ↪nesterov = True)
################################################################################
#                                 END OF YOUR CODE
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 6.1203
Checking accuracy
Got 1804 / 10195 correct (17.69)

Iteration 100, loss = 1.7327
Checking accuracy
Got 2758 / 10195 correct (27.05)

Iteration 200, loss = 1.6805
Checking accuracy
Got 2968 / 10195 correct (29.11)

Iteration 300, loss = 1.7848
Checking accuracy
Got 3264 / 10195 correct (32.02)

Iteration 400, loss = 1.5465
Checking accuracy
Got 3471 / 10195 correct (34.05)

Iteration 500, loss = 1.6186
Checking accuracy
Got 3586 / 10195 correct (35.17)

Iteration 600, loss = 1.6805
Checking accuracy
Got 3738 / 10195 correct (36.67)

Iteration 700, loss = 1.4446
```

```
Checking accuracy
Got 3817 / 10195 correct (37.44)

Iteration 800, loss = 1.4776
Checking accuracy
Got 3931 / 10195 correct (38.56)
```

7 Part V. CINIC-10 open-ended challenge

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

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves values close or above 60% accuracy on the CINIC-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

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

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

7.0.1 Things you might try:

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

7.0.2 Tips for training

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

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

7.0.3 Going above and beyond

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

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

7.0.4 Have fun and happy training!

[15]:
```
################################################################################
# TODO:                                                                        ␣
 ↪#
# Experiment with any architectures, optimizers, and hyperparameters.          #
#                                                                              #
#                                                                              #
# 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)*****

model = nn.Sequential(
    # Block 1
    nn.Conv2d(3, 64, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(64),
```

```
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(0.2),

    # Block 2
    nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(0.3),

    # Block 3
    nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(256),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(0.4),

    nn.Flatten(),
    nn.Linear(256*4*4, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 6)
)

optimizer = optim.Adam(model.parameters(), lr=1e-3)

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


train_part34(model, optimizer, epochs=10)
```

```
Iteration 0, loss = 1.9867
Checking accuracy
Got 1699 / 10195 correct (16.67)

Iteration 100, loss = 1.6358
Checking accuracy
Got 3464 / 10195 correct (33.98)

Iteration 200, loss = 1.5246
Checking accuracy
Got 3663 / 10195 correct (35.93)

Iteration 300, loss = 1.6500
```

```
Checking accuracy
Got 3931 / 10195 correct (38.56)

Iteration 400, loss = 1.5123
Checking accuracy
Got 4007 / 10195 correct (39.30)

Iteration 500, loss = 1.4281
Checking accuracy
Got 4495 / 10195 correct (44.09)

Iteration 600, loss = 1.5404
Checking accuracy
Got 4333 / 10195 correct (42.50)

Iteration 700, loss = 1.5324
Checking accuracy
Got 4492 / 10195 correct (44.06)

Iteration 800, loss = 1.3197
Checking accuracy
Got 4666 / 10195 correct (45.77)

Iteration 0, loss = 1.3318
Checking accuracy
Got 4550 / 10195 correct (44.63)

Iteration 100, loss = 1.3004
Checking accuracy
Got 4805 / 10195 correct (47.13)

Iteration 200, loss = 1.3913
Checking accuracy
Got 4898 / 10195 correct (48.04)

Iteration 300, loss = 1.4642
Checking accuracy
Got 4892 / 10195 correct (47.98)

Iteration 400, loss = 1.5159
Checking accuracy
Got 4795 / 10195 correct (47.03)

Iteration 500, loss = 1.3109
Checking accuracy
Got 5018 / 10195 correct (49.22)

Iteration 600, loss = 1.4002
```

```
Checking accuracy
Got 5033 / 10195 correct (49.37)

Iteration 700, loss = 1.4049
Checking accuracy
Got 5116 / 10195 correct (50.18)

Iteration 800, loss = 1.3292
Checking accuracy
Got 4942 / 10195 correct (48.47)

Iteration 0, loss = 1.3740
Checking accuracy
Got 5070 / 10195 correct (49.73)

Iteration 100, loss = 1.3656
Checking accuracy
Got 4981 / 10195 correct (48.86)

Iteration 200, loss = 1.4345
Checking accuracy
Got 5086 / 10195 correct (49.89)

Iteration 300, loss = 1.3422
Checking accuracy
Got 5084 / 10195 correct (49.87)

Iteration 400, loss = 1.3065
Checking accuracy
Got 4952 / 10195 correct (48.57)

Iteration 500, loss = 1.3143
Checking accuracy
Got 5117 / 10195 correct (50.19)

Iteration 600, loss = 1.4238
Checking accuracy
Got 5167 / 10195 correct (50.68)

Iteration 700, loss = 1.4319
Checking accuracy
Got 5349 / 10195 correct (52.47)

Iteration 800, loss = 1.2962
Checking accuracy
Got 5226 / 10195 correct (51.26)

Iteration 0, loss = 1.2348
```

```
Checking accuracy
Got 5286 / 10195 correct (51.85)

Iteration 100, loss = 1.1916
Checking accuracy
Got 5299 / 10195 correct (51.98)

Iteration 200, loss = 1.5096
Checking accuracy
Got 5419 / 10195 correct (53.15)

Iteration 300, loss = 1.2358
Checking accuracy
Got 5321 / 10195 correct (52.19)

Iteration 400, loss = 1.2785
Checking accuracy
Got 5477 / 10195 correct (53.72)

Iteration 500, loss = 1.2061
Checking accuracy
Got 5398 / 10195 correct (52.95)

Iteration 600, loss = 1.3300
Checking accuracy
Got 5412 / 10195 correct (53.08)

Iteration 700, loss = 1.2769
Checking accuracy
Got 5531 / 10195 correct (54.25)

Iteration 800, loss = 1.2991
Checking accuracy
Got 5497 / 10195 correct (53.92)

Iteration 0, loss = 1.2991
Checking accuracy
Got 5611 / 10195 correct (55.04)

Iteration 100, loss = 1.3235
Checking accuracy
Got 5479 / 10195 correct (53.74)

Iteration 200, loss = 1.1508
Checking accuracy
Got 5502 / 10195 correct (53.97)

Iteration 300, loss = 1.4613
```

```
Checking accuracy
Got 5495 / 10195 correct (53.90)

Iteration 400, loss = 1.2496
Checking accuracy
Got 5483 / 10195 correct (53.78)

Iteration 500, loss = 1.1412
Checking accuracy
Got 5737 / 10195 correct (56.27)

Iteration 600, loss = 1.4222
Checking accuracy
Got 5660 / 10195 correct (55.52)

Iteration 700, loss = 1.2871
Checking accuracy
Got 5571 / 10195 correct (54.64)

Iteration 800, loss = 1.2913
Checking accuracy
Got 5510 / 10195 correct (54.05)

Iteration 0, loss = 1.1405
Checking accuracy
Got 5692 / 10195 correct (55.83)

Iteration 100, loss = 1.1168
Checking accuracy
Got 5444 / 10195 correct (53.40)

Iteration 200, loss = 1.1868
Checking accuracy
Got 5671 / 10195 correct (55.63)

Iteration 300, loss = 1.3370
Checking accuracy
Got 5843 / 10195 correct (57.31)

Iteration 400, loss = 1.2664
Checking accuracy
Got 5428 / 10195 correct (53.24)

Iteration 500, loss = 1.2735
Checking accuracy
Got 5625 / 10195 correct (55.17)

Iteration 600, loss = 1.3764
```

```
Checking accuracy
Got 5805 / 10195 correct (56.94)

Iteration 700, loss = 1.0733
Checking accuracy
Got 5869 / 10195 correct (57.57)

Iteration 800, loss = 1.0985
Checking accuracy
Got 5847 / 10195 correct (57.35)

Iteration 0, loss = 1.2148
Checking accuracy
Got 5687 / 10195 correct (55.78)

Iteration 100, loss = 1.2093
Checking accuracy
Got 5674 / 10195 correct (55.65)

Iteration 200, loss = 1.1938
Checking accuracy
Got 5573 / 10195 correct (54.66)

Iteration 300, loss = 1.2872
Checking accuracy
Got 5868 / 10195 correct (57.56)

Iteration 400, loss = 1.2253
Checking accuracy
Got 5795 / 10195 correct (56.84)

Iteration 500, loss = 1.3354
Checking accuracy
Got 5812 / 10195 correct (57.01)

Iteration 600, loss = 1.2970
Checking accuracy
Got 5863 / 10195 correct (57.51)

Iteration 700, loss = 1.1490
Checking accuracy
Got 5925 / 10195 correct (58.12)

Iteration 800, loss = 1.1640
Checking accuracy
Got 5781 / 10195 correct (56.70)

Iteration 0, loss = 1.2292
```

```
Checking accuracy
Got 5750 / 10195 correct (56.40)

Iteration 100, loss = 1.1769
Checking accuracy
Got 5925 / 10195 correct (58.12)

Iteration 200, loss = 1.2656
Checking accuracy
Got 5884 / 10195 correct (57.71)

Iteration 300, loss = 1.1872
Checking accuracy
Got 6080 / 10195 correct (59.64)

Iteration 400, loss = 1.0726
Checking accuracy
Got 5750 / 10195 correct (56.40)

Iteration 500, loss = 1.2020
Checking accuracy
Got 5877 / 10195 correct (57.65)

Iteration 600, loss = 1.3219
Checking accuracy
Got 5936 / 10195 correct (58.22)

Iteration 700, loss = 1.1836
Checking accuracy
Got 6069 / 10195 correct (59.53)

Iteration 800, loss = 1.0120
Checking accuracy
Got 6018 / 10195 correct (59.03)

Iteration 0, loss = 1.1790
Checking accuracy
Got 5860 / 10195 correct (57.48)

Iteration 100, loss = 1.2659
Checking accuracy
Got 5918 / 10195 correct (58.05)

Iteration 200, loss = 1.2473
Checking accuracy
Got 5930 / 10195 correct (58.17)

Iteration 300, loss = 1.2857
```

```
Checking accuracy
Got 5983 / 10195 correct (58.69)

Iteration 400, loss = 1.3085
Checking accuracy
Got 5716 / 10195 correct (56.07)

Iteration 500, loss = 1.1419
Checking accuracy
Got 5984 / 10195 correct (58.70)

Iteration 600, loss = 1.3170
Checking accuracy
Got 6022 / 10195 correct (59.07)

Iteration 700, loss = 1.1586
Checking accuracy
Got 6107 / 10195 correct (59.90)

Iteration 800, loss = 1.0640
Checking accuracy
Got 5924 / 10195 correct (58.11)

Iteration 0, loss = 1.1091
Checking accuracy
Got 5949 / 10195 correct (58.35)

Iteration 100, loss = 1.0027
Checking accuracy
Got 6163 / 10195 correct (60.45)

Iteration 200, loss = 1.1818
Checking accuracy
Got 6032 / 10195 correct (59.17)

Iteration 300, loss = 1.2106
Checking accuracy
Got 6183 / 10195 correct (60.65)

Iteration 400, loss = 1.0219
Checking accuracy
Got 6024 / 10195 correct (59.09)

Iteration 500, loss = 1.0786
Checking accuracy
Got 6124 / 10195 correct (60.07)

Iteration 600, loss = 1.1229
```

```
Checking accuracy
Got 6006 / 10195 correct (58.91)

Iteration 700, loss = 1.0956
Checking accuracy
Got 6254 / 10195 correct (61.34)

Iteration 800, loss = 1.0051
Checking accuracy
Got 6190 / 10195 correct (60.72)
```

7.1 Describe what you did

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

I implemented a 3-layer convolutional neural network using the Sequential API. I have `Batch Normalization` after each convolution to stabilize training, `Dropout` at multiple stages, extra fully connected layer before output for more expressive power, and `weight decay` in Adam optimizer.

7.2 Test set – run this only once

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

[16]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy
Got 6266 / 10196 correct (61.46)
```

Q5-Tensorflow

October 5, 2025

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

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment3/'
FOLDERNAME = None
assert FOLDERNAME is not None, "[!] Enter the foldername."

# symlink to make it easier to load your files
!ln -s "/content/drive/My Drive/$FOLDERNAME" "/content/assignment2"

# 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))


%cd /content
```

1 What's this TensorFlow business?

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

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

What is it? 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.

Why?

- Our code will now run on GPUs! Much faster training. Writing your own modules to run on GPUs is beyond the scope of this class, unfortunately.
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.1 How will 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 the latest version of Tensorflow which is as of this homework version `2.2.0-rc3`. Most examples on the web today are still in 1.x, so be careful not to confuse the two when looking up documentation.

1.2 Install Tensorflow 2.0 (ONLY IF YOU ARE WORKING LOCALLY)

1. Have the latest version of Anaconda installed on your machine.
2. Create a new conda environment starting from Python 3.7. In this setup example, we'll call it `tf_20_env`.
3. Run the command: `source activate tf_20_env`
4. Then pip install TF 2.0 as described here: https://www.tensorflow.org/install

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 CINIC-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, CINIC-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CINIC-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 Part I: Preparation

First, we load the CINIC-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.

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.

[2]:
```
import os
import tensorflow as tf
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt
from ece662.data_utils import get_CINIC10_data
from ece662.pruning_helper import invert_ch

%matplotlib inline
```

[3]:
```
def load_cinic10():
  data = get_CINIC10_data()

  # Invoke the above function to get our data.
  X_train, y_train, X_val, y_val, X_test, y_test = data['X_train'],␣
 ↪data['y_train'], data['X_val'], data['y_val'], data['X_test'], data['y_test']
  X_train = X_train.astype(np.float32)
  X_train = X_train.transpose(0, 3, 2, 1).copy()
  X_val = X_val.astype(np.float32)
  X_val = X_val.transpose(0, 3, 2, 1).copy()
  X_test = X_test.astype(np.float32)
  X_test = X_test.transpose(0, 3, 2, 1).copy()

  y_train = y_train.astype(np.int32)
  y_val = y_val.astype(np.int32)
  y_test = y_test.astype(np.int32)

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

```
data = get_CINIC10_data()

# 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_cinic10()
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)
```

```
Train data shape:  (53973, 32, 32, 3)
Train labels shape:  (53973,) int32
Validation data shape:  (10195, 32, 32, 3)
Validation labels shape:  (10195,)
Test data shape:  (10196, 32, 32, 3)
Test labels shape:  (10196,)
```

[4]:
```
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)
```

[5]:
```
# 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,)
```

You can optionally **use GPU by setting the flag to True below**.

3.1 Colab Users

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

[38]:
```
# Set up some global variables
USE_GPU = False

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:  /cpu:0
```

4 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.

4.0.1 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.

4.0.2 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.

4.0.3 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.

[8]:
```
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))
```

[9]:
```
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)
```

4.0.4 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 CINIC10 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.

[10]:
```
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
```

[11]:
```
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, 6))

        # 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, 6)
```

4.0.5 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

[12]:
```
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)*****
    # First convolution + ReLU
    x = tf.nn.conv2d(x, conv_w1, strides=1, padding='SAME') + conv_b1
    x = tf.nn.relu(x)

    # Second convolution + ReLU
    x = tf.nn.conv2d(x, conv_w2, strides=1, padding='SAME') + conv_b2
    x = tf.nn.relu(x)

    # Flatten before fully connected layer
    x = tf.reshape(x, [x.shape[0], -1])

    # Fully connected layer to compute class scores
    scores = tf.matmul(x, fc_w) + fc_b


    # *****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, 6)`.

[13]:
```
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, 6))
        fc_b = tf.zeros((6,))
        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, 6)
```

4.0.6 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

[14]:
```
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
```

[15]:
```
def train_part2(model_fn, init_fn, learning_rate):
    """
    Train a model on CINIC-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)
```

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

4.0.7 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

[ ]:
```
```

[17]:
```
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)
```

4.0.8 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 CINIC-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 30% after one epoch of training.

[19]:
```
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, 6)))
    return [w1, w2]

learning_rate = 1e-4
train_part2(two_layer_fc, two_layer_fc_init, learning_rate)
```

```
Iteration 0, loss = 109.9078
Got 1850 / 10195 correct (18.15%)
```

4.0.9 Barebones TensorFlow: Train a three-layer ConvNet

We will now use TensorFlow to train a three-layer ConvNet on CINIC-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 6 classes

You don't need to do any hyperparameter tuning, but you should see validation accuracies above 30% after one epoch of training.

[22]:
```
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)*****

    conv_w1 = tf.Variable(create_matrix_with_kaiming_normal((5, 5, 3, 32)))
    conv_b1 = tf.Variable(tf.zeros(32))
```

```
    conv_w2 = tf.Variable(create_matrix_with_kaiming_normal((3, 3, 32, 16)))
    conv_b2 = tf.Variable(tf.zeros(16))

    fc_input_size = 32 * 32 * 16
    fc_w = tf.Variable(create_matrix_with_kaiming_normal((fc_input_size, 6)))
    fc_b = tf.Variable(tf.zeros(6))

    params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ################################################################################
    #                                END OF YOUR CODE                              #
    ################################################################################
    return params

learning_rate = 1e-4
train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate)
```

```
Iteration 0, loss = 84.5723
Got 1704 / 10195 correct (16.71%)
```

5 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.

5.0.1 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.

[23]:
```
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, 6
    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, 6)
```

5.0.2 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

[24]:
```
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)*****
        self.conv1 = tf.keras.layers.Conv2D(
            filters=channel_1,
            kernel_size=5,
            padding='same',
            activation=None,
            kernel_initializer='he_normal',
            bias_initializer='zeros'
        )

        self.relu1 = tf.keras.layers.ReLU()

        self.conv2 = tf.keras.layers.Conv2D(
            filters=channel_2,
            kernel_size=3,
            padding='same',
            activation=None,
            kernel_initializer='he_normal',
            bias_initializer='zeros'
        )

        self.relu2 = tf.keras.layers.ReLU()
```

```
        self.flatten = tf.keras.layers.Flatten()

        self.fc = tf.keras.layers.Dense(
            units=num_classes,
            activation=None,
            kernel_initializer='he_normal',
            bias_initializer='zeros'
        )

        self.softmax = tf.keras.layers.Softmax()


        # *****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)*****
        x = self.conv1(x)
        x = self.relu1(x)

        x = self.conv2(x)
        x = self.relu2(x)


        x = self.flatten(x)

        x = self.fc(x)

        scores = self.softmax(x)

        # *****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.

[25]:
```
def test_ThreeLayerConvNet():
    channel_1, channel_2, num_classes = 12, 8, 6
    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, 6)
```

5.0.3 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.

[32]:
```
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 CINIC-10 training set and periodically checks
    accuracy on the CINIC-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 training
    """
    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_state()
            train_accuracy.reset_state()

            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_state()
                        val_accuracy.reset_state()
                        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
```

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

We can now use the tools defined above to train a two-layer network on CINIC-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 30% after one epoch of training.

[33]:
```
hidden_size, num_classes = 4000, 6
learning_rate = 1e-4

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: 127.77142333984375, Accuracy: 7.8125, Val Loss:
85.68479919433594, Val Accuracy: 20.706228256225586
```

5.0.5 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 CINIC-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

[34]:
```
learning_rate = 1e-3
channel_1, channel_2, num_classes = 32, 16, 6

def model_init_fn():
    model = None
    ################################################################################
    # TODO: Complete the implementation of model_fn.                               #
    ################################################################################
```

```
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    # *****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)
    # *****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: 95.68975067138672, Accuracy: 25.0, Val Loss:
8267.0888671875, Val Accuracy: 16.684650421142578
```

6 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.

6.0.1 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 30% after training for one epoch.

[39]:
```
learning_rate = 1e-4

def model_init_fn():
    input_shape = (32, 32, 3)
    hidden_layer_size, num_classes = 4000, 6
    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: 92.65779113769531, Accuracy: 15.625, Val Loss:
97.97563171386719, Val Accuracy: 18.26385498046875
Iteration 100, Epoch 1, Loss: 80.08279418945312, Accuracy: 24.118192672729492,
Val Loss: 96.27534484863281, Val Accuracy: 24.139284133911133
Iteration 200, Epoch 1, Loss: 74.06292724609375, Accuracy: 25.52083396911621,
Val Loss: 53.816078186035156, Val Accuracy: 27.484058380126953
Iteration 300, Epoch 1, Loss: 70.02810668945312, Accuracy: 26.318519592285156,
Val Loss: 67.48982238769531, Val Accuracy: 25.326141357421875
Iteration 400, Epoch 1, Loss: 68.09843444824219, Accuracy: 26.753429412841797,
Val Loss: 49.72443389892578, Val Accuracy: 28.445316314697266
Iteration 500, Epoch 1, Loss: 65.37230682373047, Accuracy: 27.233034133911133,
Val Loss: 55.89813232421875, Val Accuracy: 26.051986694335938
Iteration 600, Epoch 1, Loss: 63.20640563964844, Accuracy: 27.550437927246094,
Val Loss: 58.9838981628418, Val Accuracy: 29.730260848999023
Iteration 700, Epoch 1, Loss: 61.568790435791016, Accuracy: 27.808486938476562,
Val Loss: 49.39629364013672, Val Accuracy: 30.76998519897461
Iteration 800, Epoch 1, Loss: 60.088016510009766, Accuracy: 28.089887619018555,
Val Loss: 48.96007537841797, Val Accuracy: 31.84894561767578
```

6.0.2 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.

[36]:
```
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)
```

```
844/844             14s 16ms/step -
loss: 59.8687 - sparse_categorical_accuracy: 0.2829 - val_loss: 65.5743 -
val_sparse_categorical_accuracy: 0.2365
319/319             1s 5ms/step -
loss: 65.0036 - sparse_categorical_accuracy: 0.2412
```

[36]:
```
[65.00361633300781, 0.24117301404476166]
```

6.0.3 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.

[40]:
```
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)*****
    initializer = tf.keras.initializers.VarianceScaling(scale=2.0,␣
 ↪mode='fan_in', distribution='untruncated_normal')

    model = tf.keras.Sequential([
        # First convolutional layer: 32 filters of size 5x5, padding=2
        tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=5,
            padding='same',
            activation=None,
```

```
            kernel_initializer=initializer,
            bias_initializer='zeros'
        ),
        tf.keras.layers.ReLU(),

        # Second convolutional layer: 16 filters of size 3x3, padding=1
        tf.keras.layers.Conv2D(
            filters=16,
            kernel_size=3,
            padding='same',
            activation=None,
            kernel_initializer=initializer,
            bias_initializer='zeros'
        ),
        tf.keras.layers.ReLU(),

        tf.keras.layers.Flatten(),

        tf.keras.layers.Dense(
            units=6,
            activation=None,
            kernel_initializer=initializer,
            bias_initializer='zeros'
        ),

        tf.keras.layers.Softmax()
    ])
    # *****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)*****
    optimizer = tf.keras.optimizers.SGD(
        learning_rate=learning_rate,
        momentum=0.9,
        nesterov=True
    )

    # *****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: 68.51043701171875, Accuracy: 20.3125, Val Loss:
617.9461669921875, Val Accuracy: 19.06817054748535
Iteration 100, Epoch 1, Loss: 7.942904949188232, Accuracy: 16.893564224243164,
Val Loss: 1.7916767597198486, Val Accuracy: 16.380578994750977
Iteration 200, Epoch 1, Loss: 4.882513523101807, Accuracy: 16.495647430419922,
Val Loss: 1.7914831638336182, Val Accuracy: 16.6944580078125
Iteration 300, Epoch 1, Loss: 3.8555352687835693, Accuracy: 16.569766998291016,
Val Loss: 1.790961503982544, Val Accuracy: 16.72388458251953
Iteration 400, Epoch 1, Loss: 3.340500593185425, Accuracy: 16.587438583374023,
Val Loss: 1.7898870706558228, Val Accuracy: 17.175085067749023
Iteration 500, Epoch 1, Loss: 3.0314345359802246, Accuracy: 16.566865921020508,
Val Loss: 1.7903368473052979, Val Accuracy: 16.871015548706055
Iteration 600, Epoch 1, Loss: 2.825112819671631, Accuracy: 16.545339584350586,
Val Loss: 1.790426254272461, Val Accuracy: 17.45953941345215
Iteration 700, Epoch 1, Loss: 2.6774659156799316, Accuracy: 16.60797119140625,
Val Loss: 1.7900571823120117, Val Accuracy: 17.430112838745117
Iteration 800, Epoch 1, Loss: 2.56681489944458, Accuracy: 16.688125610351562,
Val Loss: 1.7896082401275635, Val Accuracy: 17.685138702392578
```

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

[41]:
```
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)
```

```
844/844             21s 25ms/step -
loss: 142.2924 - sparse_categorical_accuracy: 0.1635 - val_loss: 1.7917 -
val_sparse_categorical_accuracy: 0.1667
319/319             2s 8ms/step -
loss: 1.7918 - sparse_categorical_accuracy: 0.1667
```

[41]:
```
[1.7917611598968506, 0.166732057929039]
```

6.1 Part V: Functional API

6.1.1 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.

[42]:
```
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, 6
    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, 6)
```

6.1.2 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.

[43]:
```
input_shape = (32, 32, 3)
hidden_size, num_classes = 4000, 6
learning_rate = 1e-4

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: 81.69221496582031, Accuracy: 21.875, Val Loss:
90.97663116455078, Val Accuracy: 18.381559371948242
Iteration 100, Epoch 1, Loss: 80.59698486328125, Accuracy: 23.901609420776367,
Val Loss: 94.61466979980469, Val Accuracy: 23.570377349853516
Iteration 200, Epoch 1, Loss: 75.28466033935547, Accuracy: 25.20211410522461,
Val Loss: 53.57415771484375, Val Accuracy: 27.621381759643555
Iteration 300, Epoch 1, Loss: 71.33097076416016, Accuracy: 26.116069793701172,
Val Loss: 69.66560363769531, Val Accuracy: 22.903385162353516
Iteration 400, Epoch 1, Loss: 69.3940200805664, Accuracy: 26.69498062133789, Val
Loss: 62.43996047973633, Val Accuracy: 26.463953018188477
Iteration 500, Epoch 1, Loss: 67.17097473144531, Accuracy: 26.971057891845703,
Val Loss: 67.72100830078125, Val Accuracy: 26.738595962524414
Iteration 600, Epoch 1, Loss: 64.83759307861328, Accuracy: 27.397045135498047,
Val Loss: 59.63288497924805, Val Accuracy: 29.15154457092285
Iteration 700, Epoch 1, Loss: 62.827205657958984, Accuracy: 27.70818519592285,
Val Loss: 47.71607208251953, Val Accuracy: 31.554683685302734
Iteration 800, Epoch 1, Loss: 61.278411865234375, Accuracy: 28.025514602661133,
Val Loss: 47.70978546142578, Val Accuracy: 30.975967407226562
```

7 Part V: CINIC-10 open-ended challenge

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

You should experiment with architectures, hyperparameters, loss functions, regularization, or anything else you can think of to train a model that achieves an accuracy close to 60% or above 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.

7.0.1 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?

7.0.2 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

7.0.3 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.

7.0.4 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

7.0.5 Have fun and happy training!

[44]:
```
class CustomConvNet(tf.keras.Model):
    def __init__(self):
        super(CustomConvNet, self).__init__()
        ␣
↪#########################################################################
        # TODO: Construct a model that performs well on CINIC-10
↪    #
        ␣
↪#########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


        initializer = tf.keras.initializers.HeNormal()


        self.conv1a = tf.keras.layers.Conv2D(32, 3, padding='same',
                                             kernel_initializer=initializer)
        self.bn1a = tf.keras.layers.BatchNormalization()
        self.conv1b = tf.keras.layers.Conv2D(32, 3, padding='same',
                                             kernel_initializer=initializer)
        self.bn1b = tf.keras.layers.BatchNormalization()
        self.pool1 = tf.keras.layers.MaxPooling2D(2, 2)
        self.dropout1 = tf.keras.layers.Dropout(0.25)

        self.conv2a = tf.keras.layers.Conv2D(64, 3, padding='same',
                                             kernel_initializer=initializer)
        self.bn2a = tf.keras.layers.BatchNormalization()
        self.conv2b = tf.keras.layers.Conv2D(64, 3, padding='same',
                                             kernel_initializer=initializer)
        self.bn2b = tf.keras.layers.BatchNormalization()
        self.pool2 = tf.keras.layers.MaxPooling2D(2, 2)
        self.dropout2 = tf.keras.layers.Dropout(0.25)

        self.conv3a = tf.keras.layers.Conv2D(128, 3, padding='same',
                                             kernel_initializer=initializer)
        self.bn3a = tf.keras.layers.BatchNormalization()
        self.conv3b = tf.keras.layers.Conv2D(128, 3, padding='same',
                                             kernel_initializer=initializer)
        self.bn3b = tf.keras.layers.BatchNormalization()
        self.pool3 = tf.keras.layers.MaxPooling2D(2, 2)
        self.dropout3 = tf.keras.layers.Dropout(0.25)

        self.global_pool = tf.keras.layers.GlobalAveragePooling2D()

        self.dense1 = tf.keras.layers.Dense(256, kernel_initializer=initializer)
```

```
        self.bn_dense = tf.keras.layers.BatchNormalization()
        self.dropout_dense = tf.keras.layers.Dropout(0.5)

        # Output layer
        self.output_layer = tf.keras.layers.Dense(6,␣
↪kernel_initializer=initializer)

        # *****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 CINIC-10                       ␣
↪    #
      ␣
↪#########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        x = input_tensor

        # Block 1
        x = self.conv1a(x)
        x = self.bn1a(x, training=training)
        x = tf.nn.relu(x)
        x = self.conv1b(x)
        x = self.bn1b(x, training=training)
        x = tf.nn.relu(x)
        x = self.pool1(x)
        x = self.dropout1(x, training=training)

        # Block 2
        x = self.conv2a(x)
        x = self.bn2a(x, training=training)
        x = tf.nn.relu(x)
        x = self.conv2b(x)
        x = self.bn2b(x, training=training)
        x = tf.nn.relu(x)
        x = self.pool2(x)
        x = self.dropout2(x, training=training)

        # Block 3
```

```
        x = self.conv3a(x)
        x = self.bn3a(x, training=training)
        x = tf.nn.relu(x)
        x = self.conv3b(x)
        x = self.bn3b(x, training=training)
        x = tf.nn.relu(x)
        x = self.pool3(x)
        x = self.dropout3(x, training=training)

        # Global pooling and dense layers
        x = self.global_pool(x)
        x = self.dense1(x)
        x = self.bn_dense(x, training=training)
        x = tf.nn.relu(x)
        x = self.dropout_dense(x, training=training)

        x = self.output_layer(x)



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


#device = '/device:GPU:0'
print_every = 700
num_epochs = 10

model = CustomConvNet()

def model_init_fn():
    return CustomConvNet()

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

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

```
Iteration 0, Epoch 1, Loss: 9.779241561889648, Accuracy: 15.625, Val Loss:
```

```
13.313051223754883, Val Accuracy: 16.674840927124023
Iteration 700, Epoch 1, Loss: 4.619696617126465, Accuracy: 18.53601837158203,
Val Loss: 3.508979082107544, Val Accuracy: 16.5277099609375
Iteration 1400, Epoch 2, Loss: 2.89675235748291, Accuracy: 17.97856903076172,
Val Loss: 2.103235960006714, Val Accuracy: 15.566454887390137
Iteration 2100, Epoch 3, Loss: 2.3918910026550293, Accuracy: 18.220338821411133,
Val Loss: 2.1044483184814453, Val Accuracy: 17.361452102661133
Iteration 2800, Epoch 4, Loss: 3.103733777999878, Accuracy: 16.42658042907715,
Val Loss: 2.4166622161865234, Val Accuracy: 17.3320255279541
Iteration 3500, Epoch 5, Loss: 1.852010726928711, Accuracy: 17.087499618530273,
Val Loss: 1.791760802268982, Val Accuracy: 16.77292823791504
Iteration 4200, Epoch 5, Loss: 1.8241465091705322, Accuracy: 17.100378036499023,
Val Loss: 1.791760802268982, Val Accuracy: 16.606178283691406
Iteration 4900, Epoch 6, Loss: 1.804736614227295, Accuracy: 16.916757583618164,
Val Loss: 1.791760802268982, Val Accuracy: 15.860715866088867
Iteration 5600, Epoch 7, Loss: 1.7989577054977417, Accuracy: 17.001279830932617,
Val Loss: 1.791760802268982, Val Accuracy: 16.547327041625977
Iteration 6300, Epoch 8, Loss: 1.8018618822097778, Accuracy: 17.1875, Val Loss:
1.791760802268982, Val Accuracy: 16.645414352416992
Iteration 7000, Epoch 9, Loss: 1.794388771057129, Accuracy: 16.82982063293457,
Val Loss: 1.791760802268982, Val Accuracy: 16.792545318603516
Iteration 7700, Epoch 10, Loss: 1.7958394289016724, Accuracy: 16.3988094329834,
Val Loss: 1.791760802268982, Val Accuracy: 16.910249710083008
Iteration 8400, Epoch 10, Loss: 1.7945997714996338, Accuracy:
17.201086044311523, Val Loss: 1.791760802268982, Val Accuracy: 17.16527557373047
```

7.1 Describe what you did

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

I implemented a 9-layer convolutional neural network organized in 3 blocks with progressive filters ($32 \rightarrow 64 \rightarrow 128$). I have `Batch Normalization` after each convolution to stabilize training, `Dropout` at multiple stages, `MaxPooling` to reduce spatial dimensions,`Global Average Pooling` to reduce parameters, an extra fully connected layer (256 units) before output for more expressive power, and `Adam optimizer` with learning rate 1e-3.

Q6-Pruning

October 5, 2025

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

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'ece697ls/assignments/assignment3/'
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))

%cd /content
```

1 Pruning

After learning, neural networks have modified and learned a set of parameters to perform our classification task. However, such parameters are costly to maintain and do not hold the same importance.

Wouldn't it be great could optimize our resource usage by dropping less important values ? This is where pruning comes into play.

Pruning is a technique that cuts off parameters/structures from a model to increase sparcity and decrease overall model size, similar to cutting leafs or branches from bushes and trees. This process can lead to smaller memory consumption with minimal accuracy reduction. Moreover, pruning the network may also provide a speedup since there will be less operations being performed.

The pruning process can be performed during the end of an epoch of training or after training is complete. Experimenting to find out which way works the best is part of the fun !

[7]:
```
from ece662.pruning_helper import test_model, load_model
from ece662.data_utils import get_CINIC10_data
import os
```

Below we will load a pre-trained model for you to work on. If you prefer, you can save your own model from the previous Tensorflow/Pytorch task and load it here.

[8]:
```
#This code may take a while to execute as it is training a network form scratch

data = get_CINIC10_data()
mode = 'torch'#torch or tensorflow

test_data = [data['X_test'],data['y_test']]

path = f"ece662/models/{mode}.model"

model = load_model(path,mode=mode)
test_model(model,test_data,mode=mode)
```

```
Test Acc: 0.5081
```

1.1 Unstructured Pruning

Unstructured Pruning is usually related to the pruning of weights in neural networks. The general idea is to select a set of weights according to a policy and setting them up to zero.

Common policies are random weight selection or selecting the smallers weights. Unstructured Pruning can be performed in one or multiple layers within the same network.

Altough in theory Unstructured Pruning should decrease the number of operations performed during execution there should be explicit support within the framework or hardware to bypass such operations, otherwise it will just operated over zero.

1.1.1 Perform Pruning

Using the model trained in the previous step using pytorch, perform unstructured pruning in the weights of the model by removing x% of the smallest weights.

- Increment global pruning by 10% until reaching total of 80% pruned weights
- Perform inference at the end of each pruning and observe the impact into the accuracy.

Note: The percentages are related to the entire model, not per layer.

[12]:
```
################################################################################
# TODO: Perform unstructured Pruning over the trained model using 3 different
# prunning percentages.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

import torch
import torch.nn.utils.prune as prune
import copy

pruning_percentages = [0.1, 0.2, 0.3]
```

```python
for prune_percent in pruning_percentages:
    print(f"Testing {prune_percent*100:.0f}% pruning:")

    pruned_model = copy.deepcopy(model)

    # Get all parameters of model
    parameters_to_prune = []
    for name, module in pruned_model.named_modules():
        if isinstance(module, (torch.nn.Linear, torch.nn.Conv2d)):
            parameters_to_prune.append((module, 'weight'))

    prune.global_unstructured(
        parameters_to_prune,
        pruning_method=prune.L1Unstructured,
        amount=prune_percent,
    )

    # Make pruning permanent
    for module, param in parameters_to_prune:
        prune.remove(module, param)

    # Test the pruned model
    test_model(pruned_model, test_data, mode=mode)
    print()

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

```
Testing 10% pruning:
Test Acc: 0.5091

Testing 20% pruning:
Test Acc: 0.5052

Testing 30% pruning:
Test Acc: 0.5026
```

1.2 Inline Question 1:

What happened with the accuracy as the % of pruning increased ? Why was that the case?

1.3 Answer:

As the percentage of pruning increased, the accuracy of the model decreased. This happens because we are removing weights from the model, across all layers, thus removing learned pattern from the

model. This will directly decrease the accuracy of the model. For small percentage like 10%, the effect is minimal but it have compound effect as we increase and prune across all layers at 30%.

1.4 Structured Pruning

Structured Pruning consists of removing a bigger chunk of the network parameters at the same time. Instead of removing only a few weights, it is commonplace to remove entire neurons.

For example, in Convolutional Layers, removing filters can be beneficial to improve performance as it greatly decreases the amount of computation performed. However, some of these changes may affect output dimensions which may be carried over to other parts of the network. Therefore, when performing structured pruning one must always be aware of which parameters are going to be affected.

Using the previously trained model in the CINIC-10, perform Structured Prunning only in the Convolution layers of the DNN.

[13]:
```
################################################################################
# TODO: Perform unstrucuted Pruning over the trained model using 3 different
# prunning percentages.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Define pruning percentages for structured pruning
structured_pruning_percentages = [0.1, 0.2, 0.3]  # 10%, 20%, 30%

for prune_percent in structured_pruning_percentages:
    print(f"Testing {prune_percent*100:.0f}% structured pruning on conv layers:
 ↪")

    pruned_model = copy.deepcopy(model)

    # Apply structured pruning only to convolutional layers
    conv_parameters_to_prune = []
    for name, module in pruned_model.named_modules():
        if isinstance(module, torch.nn.Conv2d):
            conv_parameters_to_prune.append((module, 'weight'))

    for module, param in conv_parameters_to_prune:
        prune.ln_structured(
            module,
            name=param,
            amount=prune_percent,
            n=1,
            dim=0
        )

    for module, param in conv_parameters_to_prune:
        prune.remove(module, param)
```

```
    # Test the pruned model
    test_model(pruned_model, test_data, mode=mode)
    print()

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

```
Testing 10% structured pruning on conv layers:
Test Acc: 0.3797

Testing 20% structured pruning on conv layers:
Test Acc: 0.2745

Testing 30% structured pruning on conv layers:
Test Acc: 0.1773
```

1.5 Inline Question 2:

What is the difference between performing Structured Pruning vs Dropout ? Why would it be beneficial to perform both techniques when developing a Neural Network?

1.6 Answer:

I think the key difference between dropout and pruning is that permanence of the removal. For dropout, we temporary drop the neurons during training the help with generalization; but prune is permanent dropping from the model. Thus, dropout would be perfect for regularization, but pruning will definitely hurt accuracy as it lose capacity in exchange for faster performance during inference.