TensorFlow

March 13, 2019

0.1 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 switch over to 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, e.g., TensorFlow. This very excellent framework 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.

Acknowledgement: This exercise is adapted from Stanford CS231n.

0.2 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.

```
In [1]: import tensorflow as tf
        import numpy as np
        import math
        import timeit
        import matplotlib.pyplot as plt
        from libs.tf_layers import *
        from libs.vis_utils import *

        %matplotlib inline

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

0.3 Load Datasets

```
In [2]: from libs.data_utils import load_CIFAR10

        NUM_CLASSES = 10
        def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=10000):
            """
            Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
            it for the two-layer neural net classifier. These are the same steps as
            we used for the SVM, but condensed to a single function.
            """
            # Load the raw CIFAR-10 data
            cifar10_dir = 'cs231n/datasets/cifar-10-batches-py/'
            X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

            # Normalize the data: subtract the mean image
            mean_image = np.mean(X_train, axis=0)
            X_train -= mean_image
            X_val -= mean_image
            X_test -= mean_image

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


        # Invoke the above function to get our data.
        X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
        print('Train data shape: ', X_train.shape)
        print('Train labels shape: ', y_train.shape)
        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:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

0.4 Example Model

0.4.1 Some useful utilities

. Remember that our image data is initially N x H x W x C, where: * N is the number of datapoints (mini-batch size) * H is the height of each image in pixels * W is the height of each image in pixels * C is the number of channels (usually 3: R, G, B)

This is the right way to represent the data when we are doing something like a 2D convolution, which needs spatial understanding of where the pixels are relative to each other. When we input image data into fully connected affine layers, however, we want each data example to be represented by a single vector – it's no longer useful to segregate the different channels, rows, and columns of the data.

0.4.2 The example model itself

The first step to training your own model is defining its architecture.

Here's an example of a convolutional neural network defined in TensorFlow – try to understand what each line is doing, remembering that each layer is composed upon the previous layer. We haven't trained anything yet - that'll come next - for now, we want you to understand how everything gets set up.

In that example, you see 2D convolutional layers (Conv2d), ReLU activations, and fully-connected layers (Linear). You also see the Hinge loss (multi-class SVM) function, and the SGD optimizer being used.

Make sure you understand **why the parameters of the Linear layer are 5408 and 10**.

0.4.3 TensorFlow Details

In TensorFlow, much like in our previous notebooks, we'll first specifically initialize our variables, and then our network model.

```
In [3]: # clear old variables
        tf.reset_default_graph()

        # setup input (e.g. the data that changes every batch)
```

```
    # The first dim is None, and gets sets automatically based on batch size fed in
    X = tf.placeholder(tf.float32, [None, 32, 32, 3])
    y = tf.placeholder(tf.int64, [None])
    is_training = tf.placeholder(tf.bool)

    def simple_model(X,y):
        # define our weights (e.g. init_two_layer_convnet)

        # setup variables
        Wconv1 = tf.get_variable("Wconv1", shape=[7, 7, 3, 32])
        bconv1 = tf.get_variable("bconv1", shape=[32])
        W1 = tf.get_variable("W1", shape=[5408, 10])
        b1 = tf.get_variable("b1", shape=[10])

        # define our graph (e.g. two_layer_convnet)
        a1 = tf.nn.conv2d(X, Wconv1, strides=[1,2,2,1], padding='VALID') + bconv1
        h1 = tf.nn.relu(a1)
        h1_flat = tf.reshape(h1,[-1,5408])
        y_out = tf.matmul(h1_flat,W1) + b1
        return y_out

    y_out = simple_model(X,y)

    # define our loss
    total_loss = tf.losses.hinge_loss(tf.one_hot(y,10),logits=y_out)
    mean_loss = tf.reduce_mean(total_loss)

    # define SGD optimizer
    optimizer = tf.train.GradientDescentOptimizer(2e-4) # select optimizer and set learnin
    train_step = optimizer.minimize(mean_loss)

    # show_graph(tf.get_default_graph().as_graph_def())
```

TensorFlow supports many other layer types, loss functions, and optimizers - you will experiment with these next. Here's the official API documentation for these (if any of the parameters used above were unclear, this resource will also be helpful).

- Layers, Activations, Loss functions : https://www.tensorflow.org/api_guides/python/nn
- Optimizers: https://www.tensorflow.org/api_guides/python/train#Optimizers
- BatchNorm: https://www.tensorflow.org/api_docs/python/tf/contrib/layers/batch_norm. Note that there are few other implementations of batch normalization layers, e.g., link 1, link 2.

0.4.4 Training the model on one epoch

Define the function to train a model as following.

```
In [4]: def run_model(session, predict, loss_val, Xd, yd,
                      epochs=1, batch_size=64, print_every=100,
                      training=None, plot_losses=False):

            # have tensorflow compute accuracy
            correct_prediction = tf.equal(tf.argmax(predict,1), y)
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

            train_indicies = np.arange(Xd.shape[0])

            training_now = (training is not None)

            # setting up variables we want to compute (and optimizing)
            # if we have a training function, add that to things we compute
            variables = [mean_loss, correct_prediction, accuracy]
            if training_now:
                variables[-1] = training

            # counter
            iter_cnt = 0
            # keep track of losses
            losses = []
            for e in range(epochs):
                # shuffle indicies
                np.random.shuffle(train_indicies)
                # keep track of accuracy
                correct = 0
                # make sure we iterate over the dataset once
                for i in range(int(math.ceil(Xd.shape[0]/batch_size))):
                    # generate indicies for the batch
                    start_idx = (i*batch_size)%Xd.shape[0]
                    idx = train_indicies[start_idx:start_idx+batch_size]

                    # create a feed dictionary for this batch
                    feed_dict = {X: Xd[idx,:],
                                 y: yd[idx],
                                 is_training: training_now }
                    # get batch size
                    actual_batch_size = yd[idx].shape[0]

                    # have tensorflow compute loss and correct predictions
                    # and (if given) perform a training step
                    loss, corr, _ = session.run(variables,feed_dict=feed_dict)
                    corr = np.array(corr).astype(np.float32)

                    # aggregate performance stats
                    losses.append(loss*actual_batch_size)
                    correct += np.sum(corr)

                    # print every now and then
```

```
                if training_now and (iter_cnt % print_every) == 0:
                    print("Iteration {0}: with minibatch training loss = {1:.3g} and accura
                          .format(iter_cnt,loss,np.sum(corr)/actual_batch_size))
                iter_cnt += 1
            total_correct = correct/Xd.shape[0]
            total_loss = np.sum(losses)/Xd.shape[0]
            print("Epoch {2}, Overall loss = {0:.3g} and accuracy of {1:.3g}"\
                  .format(total_loss,total_correct,e+1))

        if plot_losses:
            plt.plot(losses)
            plt.grid(True)
            plt.title('Epoch {} Loss'.format(e+1))
            plt.xlabel('minibatch number')
            plt.ylabel('minibatch loss')
            plt.show()

        return total_loss,total_correct,losses
```

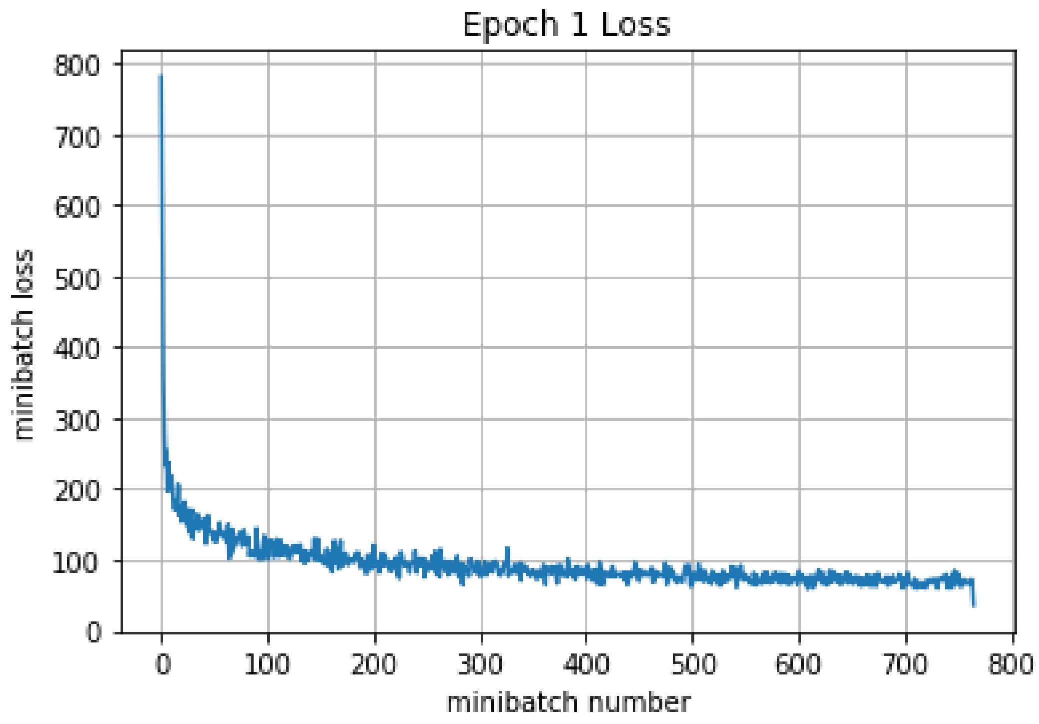
While we have defined a graph of operations above, in order to execute TensorFlow Graphs, by feeding them input data and computing the results, we first need to create a `tf.Session` object. A session encapsulates the control and state of the TensorFlow runtime. For more information, see the TensorFlow Getting started guide.

Optionally we can also specify a device context such as `/cpu:0` or `/gpu:0`. For documentation on this behavior see this TensorFlow guide. Generally, if your machine has GPU available (with all required drivers) and you install Tensorflow GPU version, Tensorflow will automatically select GPU as the primary device. Otherwise, CPU will be selected as the primary device.

You should see a validation loss of around 1 and an accuracy of 0.2 below

```
In [5]: with tf.Session() as sess:
            #with tf.device("/cpu:0"): #"/cpu:0" or "/gpu:0"
            sess.run(tf.global_variables_initializer())
            print('Training')
            run_model(sess,y_out,mean_loss,X_train,y_train,1,64,100,train_step,plot_losses=Tru
            print('Validation')
            run_model(sess,y_out,mean_loss,X_val,y_val,1,64)
```

```
Training
Iteration 0: with minibatch training loss = 12.2 and accuracy of 0.078
Iteration 100: with minibatch training loss = 1.59 and accuracy of 0.25
Iteration 200: with minibatch training loss = 1.89 and accuracy of 0.12
Iteration 300: with minibatch training loss = 1.39 and accuracy of 0.11
Iteration 400: with minibatch training loss = 1.37 and accuracy of 0.16
Iteration 500: with minibatch training loss = 1.07 and accuracy of 0.25
Iteration 600: with minibatch training loss = 1.32 and accuracy of 0.16
Iteration 700: with minibatch training loss = 0.962 and accuracy of 0.22
Epoch 1, Overall loss = 1.46 and accuracy of 0.204
```



```
Validation
Epoch 1, Overall loss = 1.07 and accuracy of 0.234
```

1 Go Deeper

In the previous exercises, you are required to implement different functions, e.g., affine, relu, conv2d, dropout, . . . which serve as basic module to build a Deep Neuron Network. Similarly, we provide the basic modules for Tensorflow in `libs/tf_layers.py`.

NOTE: In this exercise, you are welcome to change the block functions in `libs/tf_layers.py` to fit your needs the best.

```
In [6]: # define a deep network
        def deep_model(X, y, batchnorm=False, name=None):
            output = Conv2D(X, 3, 7, 8, name=name+'_conv1')
            output = tf.nn.relu(output, name=name+'_relu1')
            if batchnorm:
                output = BatchNormalization(output, True, name=name+'_BN1')
            output = Conv2D(output, 8, 7, 8, name=name+'_conv2')
            output = tf.nn.relu(output, name=name+'_relu2')
            if batchnorm:
                output = BatchNormalization(output, True, name=name+'_BN2')
            output = MaxPooling2D(output, name=name+'_maxpool1')
```

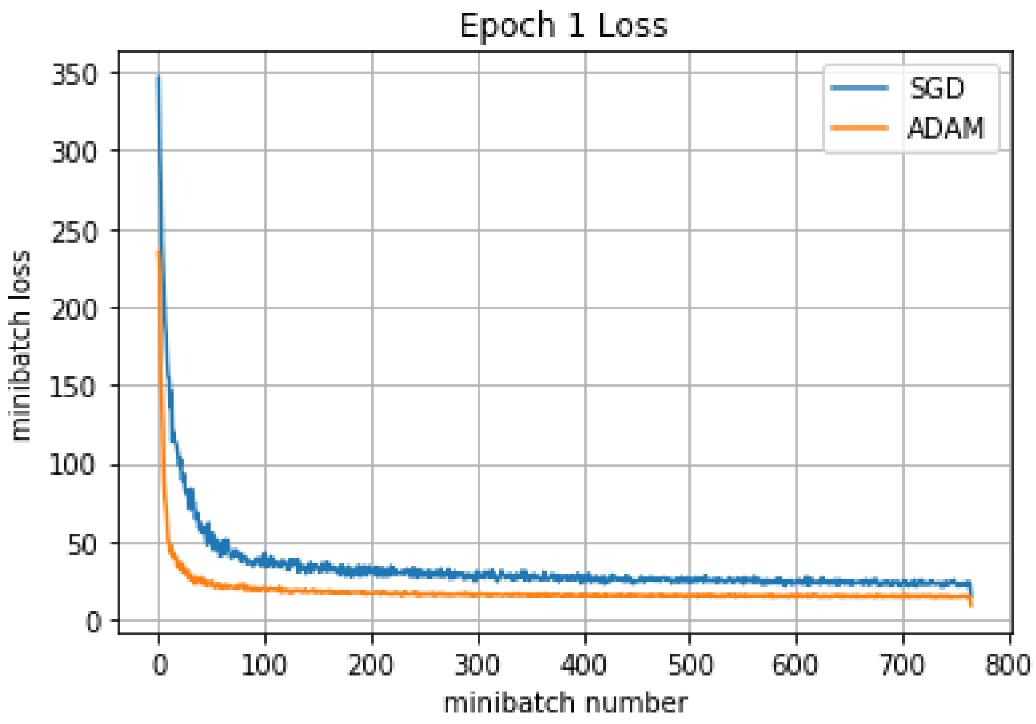
```
    output = Conv2D(output, 8, 7, 16, name=name+'_conv3')
    output = tf.nn.relu(output, name=name+'_relu3')
    if batchnorm:
        output = BatchNormalization(output, True, name=name+'_BN3')
    output = Conv2D(output, 16, 7, 16, name=name+'_conv4')

    # Here is another way of defining a name for a layer
    with tf.variable_scope(name+'_relu4'):
#         output = tf.nn.relu(output, name=name+'_relu4')
        output = tf.nn.relu(output)
    if batchnorm:
        output = BatchNormalization(output, True, name=name+'_BN4')
    output = MaxPooling2D(output, name=name+'_maxpool2')
    output = tf.reshape(output, [-1, 16*8*8], name=name+'_flatten')
    output = FullyConnected(output, 16*8*8, 100, name=name+'_fc1')
    output = tf.nn.relu(output, name=name+'_relu5')
    output = FullyConnected(output, 100, 100, name=name+'_fc2')
    output = tf.nn.relu(output, name=name+'_relu6')
    output = FullyConnected(output, 100, 10, name=name+'_fc3')
    return output
```

You are going to see ADAM optimizer being used. Now, we will compare the training loss curves of a model with SGD and ADAM optimizers.

You can try other optimizers, e.g., SGD+Momentum, RMSprop, Adagrad, Adadelta.

```
In [7]: # clear old variables
        tf.reset_default_graph()

        # setup input (e.g. the data that changes every batch)
        # The first dim is None, and gets sets automatically based on batch size fed in
        X = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int64, [None])
        is_training = tf.placeholder(tf.bool)

        y_out_deep = deep_model(X,y, False, name='deep1')

        # define our loss
        total_loss_deep = tf.losses.hinge_loss(tf.one_hot(y,10),logits=y_out_deep)
        mean_loss = tf.reduce_mean(total_loss_deep)

        # define SGD optimizer
        print('SGD optimizer')
        optimizer = tf.train.GradientDescentOptimizer(1e-4) # select optimizer and set learnin
        train_step = optimizer.minimize(mean_loss)

        with tf.Session() as sess:
            #with tf.device("/cpu:0"): #"/cpu:0" or "/gpu:0"
            sess.run(tf.global_variables_initializer())
            print('Training')
            _,_, sgd_losses = run_model(sess,y_out_deep,mean_loss,X_train,y_train,1,64,100,tra
            print('Validation')
            run_model(sess,y_out_deep,mean_loss,X_val,y_val,1,64)

        print("==========================================================\n")
        # define Adam optimizer
        print('ADAM optimizer')
        optimizer = tf.train.AdamOptimizer(1e-4) # select optimizer and set learning rate
        train_step = optimizer.minimize(mean_loss)

        with tf.Session() as sess:
        #     with tf.device("/cpu:0"): #"/cpu:0" or "/gpu:0"
            sess.run(tf.global_variables_initializer())
            print('Training')
            _,_, adam_losses = run_model(sess,y_out_deep,mean_loss,X_train,y_train,1,64,100,tra
            print('Validation')
            run_model(sess,y_out_deep,mean_loss,X_val,y_val,1,64)

        plt.plot(sgd_losses, label='SGD')
        plt.plot(adam_losses, label='ADAM')
        # plt.plot(adam_losses_batchnorm, label='ADAM+BatchNorm')
        # plt.ylim( (0, 100) )
        plt.grid(True)
        plt.legend()
        plt.title('Epoch 1 Loss')
        plt.xlabel('minibatch number')
        plt.ylabel('minibatch loss')
        plt.show()
```

```
SGD optimizer
Training
Iteration 0: with minibatch training loss = 5.41 and accuracy of 0.11
Iteration 100: with minibatch training loss = 0.664 and accuracy of 0.047
Iteration 200: with minibatch training loss = 0.51 and accuracy of 0.12
Iteration 300: with minibatch training loss = 0.483 and accuracy of 0.062
Iteration 400: with minibatch training loss = 0.439 and accuracy of 0.094
Iteration 500: with minibatch training loss = 0.379 and accuracy of 0.12
Iteration 600: with minibatch training loss = 0.383 and accuracy of 0.11
Iteration 700: with minibatch training loss = 0.371 and accuracy of 0.11
Epoch 1, Overall loss = 0.525 and accuracy of 0.108
Validation
Epoch 1, Overall loss = 0.352 and accuracy of 0.12
==========================================================

ADAM optimizer
Training
Iteration 0: with minibatch training loss = 3.67 and accuracy of 0.094
Iteration 100: with minibatch training loss = 0.306 and accuracy of 0.19
Iteration 200: with minibatch training loss = 0.266 and accuracy of 0.16
Iteration 300: with minibatch training loss = 0.249 and accuracy of 0.16
Iteration 400: with minibatch training loss = 0.257 and accuracy of 0.12
Iteration 500: with minibatch training loss = 0.237 and accuracy of 0.17
Iteration 600: with minibatch training loss = 0.239 and accuracy of 0.2
Iteration 700: with minibatch training loss = 0.233 and accuracy of 0.2
Epoch 1, Overall loss = 0.288 and accuracy of 0.191
Validation
Epoch 1, Overall loss = 0.232 and accuracy of 0.243
```

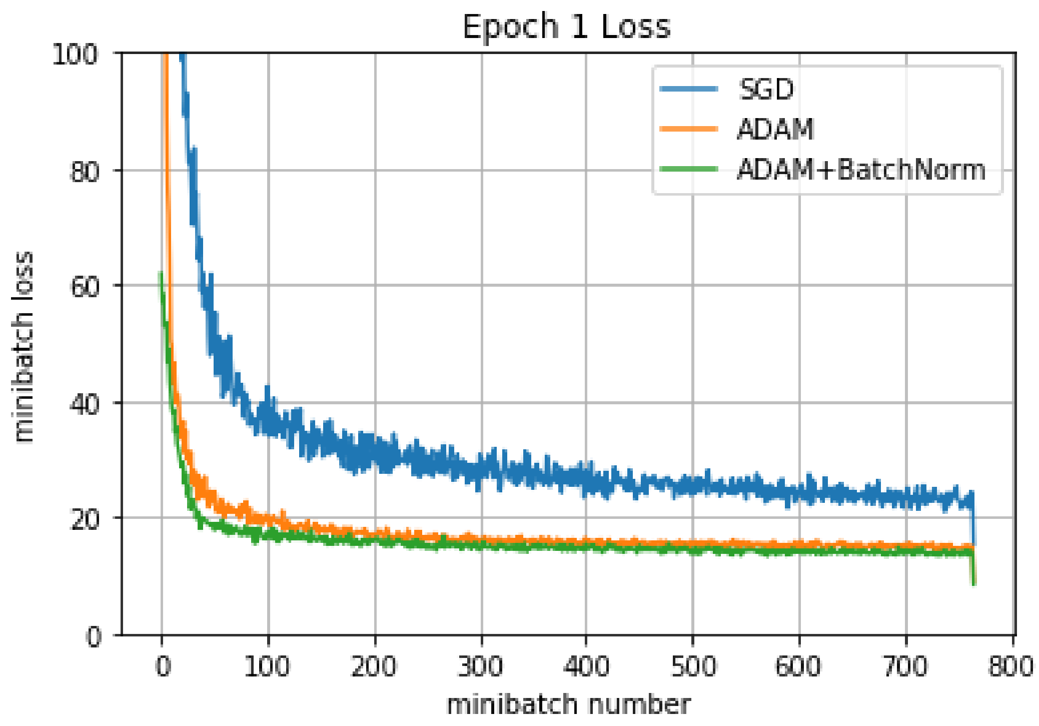


We will now see the benefit of using **batch normalization** layer.

```
In [8]: # clear old variables
        tf.reset_default_graph()

        # setup input (e.g. the data that changes every batch)
        # The first dim is None, and gets sets automatically based on batch size fed in
        X = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int64, [None])
        is_training = tf.placeholder(tf.bool)

        y_out_deep2 = deep_model(X,y, True, name='deep2')
        # define our loss
```

```python
total_loss_deep2 = tf.losses.hinge_loss(tf.one_hot(y,10),logits=y_out_deep2)
mean_loss = tf.reduce_mean(total_loss_deep2)

print("==========================================================\n")
# define Adam optimizer
print('ADAM optimizer')
optimizer = tf.train.AdamOptimizer(1e-4) # select optimizer and set learning rate

# batch normalization in tensorflow requires this extra dependency
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(extra_update_ops):
    train_step = optimizer.minimize(mean_loss)

with tf.Session() as sess:
#     with tf.device("/cpu:0"): #"/cpu:0" or "/gpu:0"
    sess.run(tf.global_variables_initializer())
    print('Training')
    _,_, adam_losses_batchnorm = run_model(sess,y_out_deep2,mean_loss,X_train,y_train,
    print('Validation')
    run_model(sess,y_out_deep2,mean_loss,X_val,y_val,1,64)

plt.plot(sgd_losses, label='SGD')
plt.plot(adam_losses, label='ADAM')
plt.plot(adam_losses_batchnorm, label='ADAM+BatchNorm')
plt.ylim( (0, 100) )
plt.grid(True)
plt.legend()
plt.title('Epoch 1 Loss')
plt.xlabel('minibatch number')
plt.ylabel('minibatch loss')
plt.show()
```

```
==========================================================

ADAM optimizer
Training
Iteration 0: with minibatch training loss = 0.968 and accuracy of 0.094
Iteration 100: with minibatch training loss = 0.273 and accuracy of 0.25
Iteration 200: with minibatch training loss = 0.249 and accuracy of 0.2
Iteration 300: with minibatch training loss = 0.233 and accuracy of 0.28
Iteration 400: with minibatch training loss = 0.241 and accuracy of 0.14
Iteration 500: with minibatch training loss = 0.229 and accuracy of 0.28
Iteration 600: with minibatch training loss = 0.225 and accuracy of 0.34
Iteration 700: with minibatch training loss = 0.213 and accuracy of 0.31
Epoch 1, Overall loss = 0.25 and accuracy of 0.261
Validation
Epoch 1, Overall loss = 0.216 and accuracy of 0.33
```

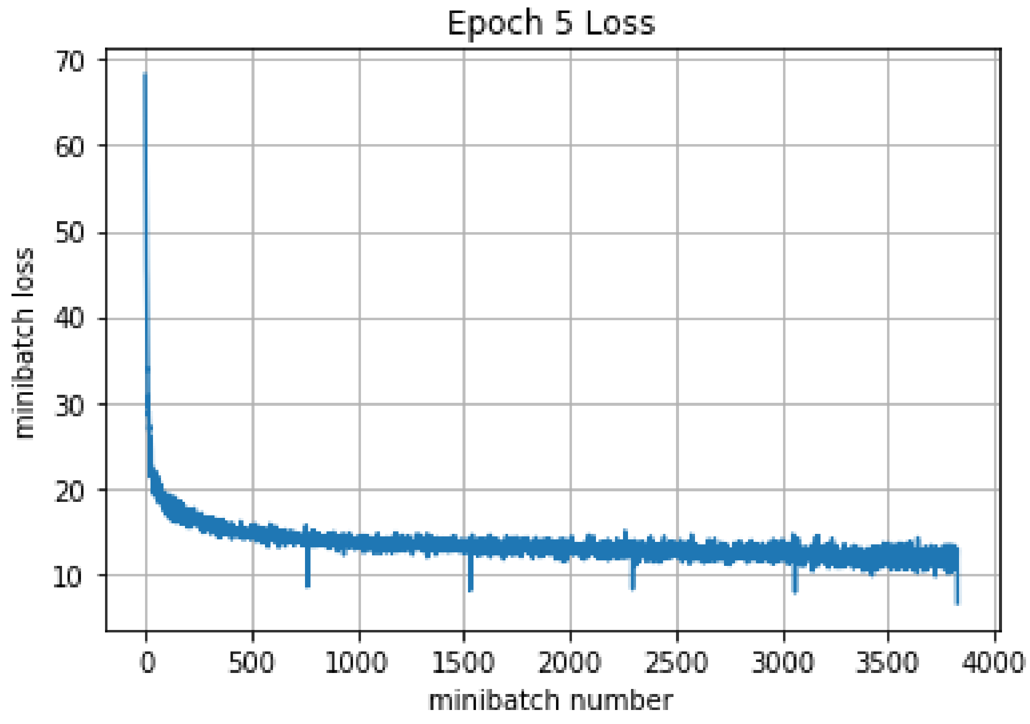


1.0.1 More epochs

Train the model with more epochs to see how good performance it can achieve.
NOTE: If you run this on a CPU, it will take some time.

```
In [9]: # clear old variables
        tf.reset_default_graph()

        # setup input (e.g. the data that changes every batch)
        # The first dim is None, and gets sets automatically based on batch size fed in
        X = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int64, [None])
        is_training = tf.placeholder(tf.bool)

        y_out_deep2 = deep_model(X,y, True, name='deep2')
        # define our loss
        total_loss_deep2 = tf.losses.hinge_loss(tf.one_hot(y,10),logits=y_out_deep2)
        mean_loss = tf.reduce_mean(total_loss_deep2)

        print("============================================================")
        # define Adam optimizer
        print('ADAM optimizer')
        optimizer = tf.train.AdamOptimizer(1e-4) # select optimizer and set learning rate
        # batch normalization in tensorflow requires this extra dependency
        extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(extra_update_ops):
            train_step = optimizer.minimize(mean_loss)

        with tf.Session() as sess:
        #     with tf.device("/cpu:0"): #"/cpu:0" or "/gpu:0"
            sess.run(tf.global_variables_initializer())
            print('Training')
            _,_, adam_losses_batchnorm = run_model(sess,y_out_deep2,mean_loss,X_train,y_train,
            print('Validation')
            run_model(sess,y_out_deep2,mean_loss,X_val,y_val,1,64)
```

```
==========================================================
ADAM optimizer
Training
Iteration 0: with minibatch training loss = 1.07 and accuracy of 0.11
Iteration 100: with minibatch training loss = 0.264 and accuracy of 0.27
Iteration 200: with minibatch training loss = 0.261 and accuracy of 0.23
Iteration 300: with minibatch training loss = 0.241 and accuracy of 0.3
Iteration 400: with minibatch training loss = 0.234 and accuracy of 0.42
Iteration 500: with minibatch training loss = 0.232 and accuracy of 0.34
Iteration 600: with minibatch training loss = 0.225 and accuracy of 0.3
Iteration 700: with minibatch training loss = 0.226 and accuracy of 0.38
Epoch 1, Overall loss = 0.258 and accuracy of 0.285
Iteration 800: with minibatch training loss = 0.207 and accuracy of 0.42
Iteration 900: with minibatch training loss = 0.217 and accuracy of 0.39
Iteration 1000: with minibatch training loss = 0.217 and accuracy of 0.33
Iteration 1100: with minibatch training loss = 0.214 and accuracy of 0.42
Iteration 1200: with minibatch training loss = 0.208 and accuracy of 0.36
Iteration 1300: with minibatch training loss = 0.226 and accuracy of 0.3
Iteration 1400: with minibatch training loss = 0.221 and accuracy of 0.33
Iteration 1500: with minibatch training loss = 0.211 and accuracy of 0.39
Epoch 2, Overall loss = 0.47 and accuracy of 0.388
Iteration 1600: with minibatch training loss = 0.219 and accuracy of 0.28
Iteration 1700: with minibatch training loss = 0.213 and accuracy of 0.36
Iteration 1800: with minibatch training loss = 0.208 and accuracy of 0.41
Iteration 1900: with minibatch training loss = 0.217 and accuracy of 0.33
Iteration 2000: with minibatch training loss = 0.207 and accuracy of 0.41
Iteration 2100: with minibatch training loss = 0.197 and accuracy of 0.41
Iteration 2200: with minibatch training loss = 0.205 and accuracy of 0.45
Epoch 3, Overall loss = 0.672 and accuracy of 0.434
Iteration 2300: with minibatch training loss = 0.21 and accuracy of 0.41
Iteration 2400: with minibatch training loss = 0.199 and accuracy of 0.47
Iteration 2500: with minibatch training loss = 0.204 and accuracy of 0.44
Iteration 2600: with minibatch training loss = 0.19 and accuracy of 0.47
Iteration 2700: with minibatch training loss = 0.204 and accuracy of 0.52
Iteration 2800: with minibatch training loss = 0.189 and accuracy of 0.52
Iteration 2900: with minibatch training loss = 0.19 and accuracy of 0.48
Iteration 3000: with minibatch training loss = 0.19 and accuracy of 0.61
Epoch 4, Overall loss = 0.868 and accuracy of 0.469
Iteration 3100: with minibatch training loss = 0.195 and accuracy of 0.52
Iteration 3200: with minibatch training loss = 0.178 and accuracy of 0.56
Iteration 3300: with minibatch training loss = 0.171 and accuracy of 0.53
Iteration 3400: with minibatch training loss = 0.165 and accuracy of 0.55
Iteration 3500: with minibatch training loss = 0.189 and accuracy of 0.41
Iteration 3600: with minibatch training loss = 0.214 and accuracy of 0.39
Iteration 3700: with minibatch training loss = 0.17 and accuracy of 0.61
Iteration 3800: with minibatch training loss = 0.185 and accuracy of 0.53
Epoch 5, Overall loss = 1.05 and accuracy of 0.501
```



```
Validation
Epoch 1, Overall loss = 0.181 and accuracy of 0.502
```

1.1 Graph visualization

```
In [10]: def variable_summaries(var):
             with tf.name_scope('summaries'):
                 mean = tf.reduce_mean(var)
                 tf.summary.scalar('mean', mean)
                 with tf.name_scope('stddev'):
```

```
                stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
            tf.summary.scalar('stddev', stddev)
            tf.summary.scalar('max', tf.reduce_max(var))
            tf.summary.scalar('min', tf.reduce_min(var))
            tf.summary.histogram('histogram', var)
```

```
In [11]: # clear old variables
         tf.reset_default_graph()

         # setup input (e.g. the data that changes every batch)
         # The first dim is None, and gets sets automatically based on batch size fed in
         X = tf.placeholder(tf.float32, [None, 32, 32, 3])
         y = tf.placeholder(tf.int64, [None])
         is_training = tf.placeholder(tf.bool)

         def simple_model(X,y):
             # define our weights (e.g. init_two_layer_convnet)

             # setup variables
             Wconv1 = tf.get_variable("Wconv1", shape=[7, 7, 3, 32])
             bconv1 = tf.get_variable("bconv1", shape=[32])
             W1 = tf.get_variable("W1", shape=[5408, 10])
             b1 = tf.get_variable("b1", shape=[10])
             variable_summaries(Wconv1)
             variable_summaries(W1)

             # define our graph (e.g. two_layer_convnet)
             a1 = tf.nn.conv2d(X, Wconv1, strides=[1,2,2,1], padding='VALID') + bconv1
             h1 = tf.nn.relu(a1)
             h1_flat = tf.reshape(h1,[-1,5408])
             y_out = tf.matmul(h1_flat,W1) + b1
             return y_out

         y_out = simple_model(X,y)

         # define our loss
         total_loss = tf.losses.hinge_loss(tf.one_hot(y,10),logits=y_out)
         mean_loss = tf.reduce_mean(total_loss)

         # define SGD optimizer
         optimizer = tf.train.GradientDescentOptimizer(2e-4) # select optimizer and set learni
         train_step = optimizer.minimize(mean_loss)
```

```
In [12]: show_graph(tf.get_default_graph().as_graph_def())
```

```
<IPython.core.display.HTML object>
```

```
In [13]: def run_model_with_tensorboard(session, predict, loss_value, Xd, yd,
                       epochs=1, batch_size=64, print_every=100,
                       training=None, tensorboard_writer=None):

             # have tensorflow compute accuracy
             correct_prediction = tf.equal(tf.argmax(predict,1), y)
             accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

             train_indicies = np.arange(Xd.shape[0])

             training_now = (training is not None)

             tf.summary.scalar("cost", loss_value)
             tf.summary.scalar("accuracy", accuracy)
             summary_op = tf.summary.merge_all()

             # counter
             iter_cnt = 0
             # keep track of losses
             losses = []
             for e in range(epochs):
                 # shuffle indicies
                 np.random.shuffle(train_indicies)
                 # keep track of accuracy
                 correct = 0
                 # make sure we iterate over the dataset once
                 batch_count = int(math.ceil(Xd.shape[0]/batch_size))
                 for i in range(batch_count):
                     # generate indicies for the batch
                     start_idx = (i*batch_size)%Xd.shape[0]
                     idx = train_indicies[start_idx:start_idx+batch_size]

                     # create a feed dictionary for this batch
                     feed_dict = {X: Xd[idx,:],
                                  y: yd[idx],
                                  is_training: training_now }

                     # get batch size
                     actual_batch_size = yd[idx].shape[0]

                     # have tensorflow compute loss and correct predictions
                     # and (if given) perform a training step
                     if training_now:
                         _, summary = session.run([training, summary_op],feed_dict=feed_dict)
                         # write log
                         tensorboard_writer.add_summary(summary, e * batch_count + i)
                     else:
                         summary = session.run(summary_op, feed_dict=feed_dict)
             return
```

1.1.1 Tensorboard for Visualization

Tensorflow provides a very useful tool: Tensorboard. This is very helpful to visualize the training loss, accuray, filters,...

Here is a good video about Tensorboard: https://www.youtube.com/watch?v=eBbEDRsCmv4

```
In [14]: # define SGD optimizer
         print('SGD optimizer')
         optimizer = tf.train.AdamOptimizer(1e-4) # select optimizer and set learning rate
         train_step = optimizer.minimize(mean_loss)

         with tf.Session() as sess:
             train_writer = tf.summary.FileWriter('logs/train', graph=tf.get_default_graph())
             #with tf.device("/cpu:0"): #"/cpu:0" or "/gpu:0"
             sess.run(tf.global_variables_initializer())
             print('Training')
             run_model_with_tensorboard(sess,y_out,mean_loss,X_train,y_train,1,64,100,train_st
                                        tensorboard_writer=train_writer)

         # tensorboard --logdir=logs/train
         print("Run the command line:\n" \
                   "--> tensorboard --logdir=logs/train " \
                   "\nThen open http://0.0.0.0:6006/ into your web browser")
```

```
SGD optimizer
Training
Run the command line:
--> tensorboard --logdir=logs/train
Then open http://0.0.0.0:6006/ into your web browser
```

1.2 Train a *great* model on CIFAR-10!

1.2.1 Things you should try:

- **Filter size**: Above we used 7x7; this makes pretty pictures but smaller filters may 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]
- **Use TensorFlow Scope**: Use TensorFlow scope and/or tf.layers to make it easier to write deeper networks. See this tutorial for how to use `tf.layers`.

- **Use Learning Rate Decay**: As the notes point out, decaying the learning rate might help the model converge. Feel free to decay every epoch, when loss doesn't change over an entire epoch, or any other heuristic you find appropriate. See the Tensorflow documentation for learning rate decay.
- **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 as in the TensorFlow MNIST tutorial.

NOTE: * In this exercise, you are welcome to change the block functions in `libs/tf_layers.py` to fit your needs the best. * Softmax cross-entropy loss: tf.losses.softmax_cross_entropy * SVM loss: tf.losses.hinge_loss

1.2.2 Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. 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 we'll save the test set for evaluating your architecture on the best parameters as selected by the validation set.

1.2.3 What we expect

At the very least, you should be able to train a ConvNet that gets at **>= 60% accuracy on the validation set**. This is just a lower bound - if you are careful it should be possible to get accuracies much higher than that! Extra credit points will be awarded for particularly high-scoring models or unique approaches.

You should use the space below to experiment and train your network. The final cell in this notebook should contain the training and validation set accuracies for your final trained network.

Have fun and happy training!

1.2.4 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; however they would be good things to try for extra credit.

- Alternative update steps: For the assignment we implemented SGD+momentum, RMSprop, and Adam; you could try alternatives like AdaGrad or AdaDelta.

- 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

If you do decide to implement something extra, clearly describe it in the "Extra Credit Description" cell below.

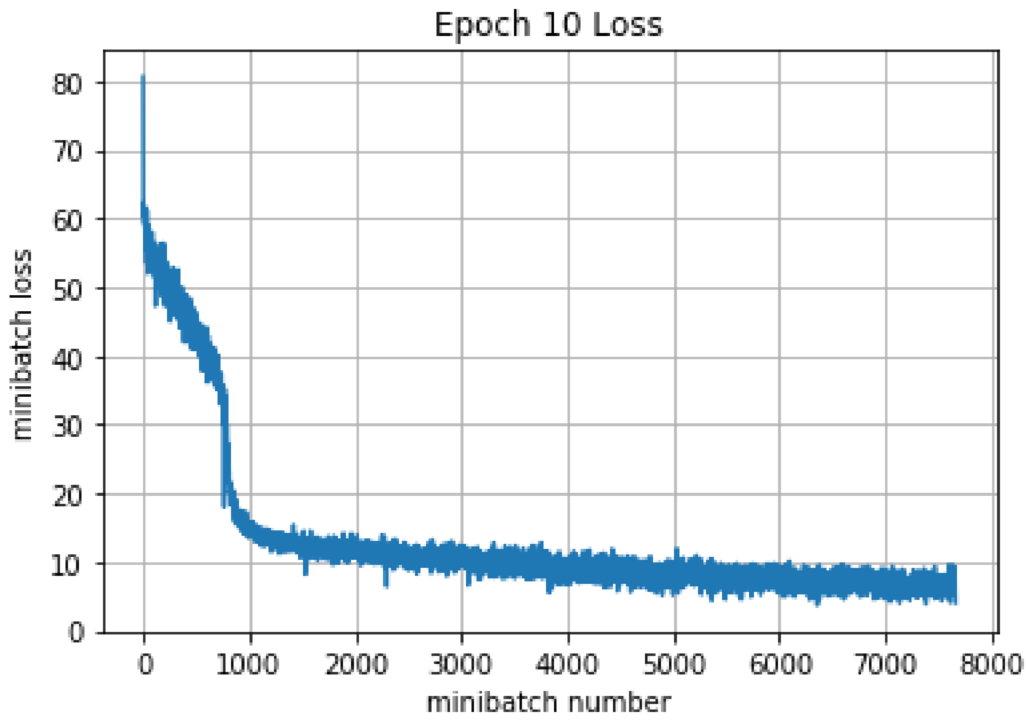
```
In [33]: # Feel free to play with this cell

         def my_model(X,y,is_training):
             # setup variables

             W1 = tf.get_variable("W1", shape=[10, 10])
             b1 = tf.get_variable("b1", shape=[10])
             variable_summaries(W1)

             # define our graph (e.g. two_layer_convnet)
             feature0 = tf.layers.conv2d(X,64,3)
             feature0r = tf.nn.relu(feature0)
             feature1 = tf.layers.conv2d(feature0r,64,3)
             feature1r = tf.nn.relu(feature1)
             feature2 = tf.layers.max_pooling2d(feature1r,2,2, padding='same')
             feature3 = tf.layers.conv2d(feature2,128,3)
             feature3r = tf.nn.relu(feature3)
             feature4 = tf.layers.conv2d(feature3r,128,3)
             feature4r = tf.nn.relu(feature4)
             feature5 = tf.layers.max_pooling2d(feature4r,2,2, padding='same')
             feature6 = tf.layers.conv2d(feature5,256,3)
             feature6r = tf.nn.relu(feature6)
             feature7 = tf.layers.conv2d(feature6r,256,3)
             feature7r = tf.nn.relu(feature7)
             feature8 = tf.layers.max_pooling2d(feature7r,2,2, padding='same')
         # Adapted VGG
         #     feature9 = tf.layers.conv2d(feature8,512,3)
         #     feature9r = tf.nn.relu(feature9)
         #     feature10 = tf.layers.conv2d(feature9r,512,3)
         #     feature10r = tf.nn.relu(feature10)
         #     feature11 = tf.layers.max_pooling2d(feature10r,2,2, padding='same')
         #     feature12 = tf.layers.conv2d(feature11,512,3)
         #     feature12r = tf.nn.relu(feature12)
         #     feature13 = tf.layers.conv2d(feature12r,512,3)
         #     feature13r = tf.nn.relu(feature13)
         #     feature14 = tf.layers.max_pooling2d(feature13r,2,2, padding='same')
```

```
            layer0 = tf.layers.average_pooling2d(feature8, 7, 1, padding='same')
            layer1 = tf.contrib.layers.fully_connected(layer0, 4096)
            layer2 = tf.nn.relu(layer1)
            layer3 = tf.nn.dropout(layer2, 0.5)
            layer4 = tf.contrib.layers.fully_connected(layer3, 4096)
            layer5 = tf.nn.relu(layer4)
            layer6 = tf.nn.dropout(layer5, 0.5)
            layer7 = tf.contrib.layers.fully_connected(layer6, 10)
            flat = tf.reshape(layer7,[-1,10])
            y_out = tf.matmul(flat,W1) + b1
            return y_out

        tf.reset_default_graph()

        X = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int64, [None])
        is_training = tf.placeholder(tf.bool)

        y_out = my_model(X,y,is_training)
        total_loss = tf.losses.hinge_loss(tf.one_hot(y,10),logits=y_out)
        mean_loss = tf.reduce_mean(total_loss)
        optimizer = tf.train.AdamOptimizer(1e-4)
        train_step = optimizer.minimize(mean_loss)
```

```
In [34]: # Feel free to play with this cell
         # This default code creates a session
         # and trains your model for 10 epochs
         # then prints the validation set accuracy
         sess = tf.Session()

         sess.run(tf.global_variables_initializer())
         print('Training')
         run_model(sess,y_out,mean_loss,X_train,y_train,10,64,100,train_step,True)
         print('Validation')
         run_model(sess,y_out,mean_loss,X_val,y_val,1,64)
```

```
Training
Iteration 0: with minibatch training loss = 1.26 and accuracy of 0.078
Iteration 100: with minibatch training loss = 0.819 and accuracy of 0.19
Iteration 200: with minibatch training loss = 0.788 and accuracy of 0.094
Iteration 300: with minibatch training loss = 0.781 and accuracy of 0.062
Iteration 400: with minibatch training loss = 0.731 and accuracy of 0.047
Iteration 500: with minibatch training loss = 0.669 and accuracy of 0.11
Iteration 600: with minibatch training loss = 0.65 and accuracy of 0.062
Iteration 700: with minibatch training loss = 0.564 and accuracy of 0.11
Epoch 1, Overall loss = 0.728 and accuracy of 0.102
```

```
Iteration 800: with minibatch training loss = 0.36 and accuracy of 0.23
Iteration 900: with minibatch training loss = 0.259 and accuracy of 0.28
Iteration 1000: with minibatch training loss = 0.228 and accuracy of 0.31
Iteration 1100: with minibatch training loss = 0.22 and accuracy of 0.33
Iteration 1200: with minibatch training loss = 0.205 and accuracy of 0.34
Iteration 1300: with minibatch training loss = 0.207 and accuracy of 0.41
Iteration 1400: with minibatch training loss = 0.203 and accuracy of 0.27
Iteration 1500: with minibatch training loss = 0.217 and accuracy of 0.28
Epoch 2, Overall loss = 0.961 and accuracy of 0.341
Iteration 1600: with minibatch training loss = 0.172 and accuracy of 0.48
Iteration 1700: with minibatch training loss = 0.18 and accuracy of 0.5
Iteration 1800: with minibatch training loss = 0.198 and accuracy of 0.34
Iteration 1900: with minibatch training loss = 0.176 and accuracy of 0.45
Iteration 2000: with minibatch training loss = 0.196 and accuracy of 0.41
Iteration 2100: with minibatch training loss = 0.165 and accuracy of 0.5
Iteration 2200: with minibatch training loss = 0.169 and accuracy of 0.52
Epoch 3, Overall loss = 1.14 and accuracy of 0.44
Iteration 2300: with minibatch training loss = 0.181 and accuracy of 0.52
Iteration 2400: with minibatch training loss = 0.18 and accuracy of 0.53
Iteration 2500: with minibatch training loss = 0.137 and accuracy of 0.64
Iteration 2600: with minibatch training loss = 0.159 and accuracy of 0.55
Iteration 2700: with minibatch training loss = 0.152 and accuracy of 0.59
Iteration 2800: with minibatch training loss = 0.15 and accuracy of 0.53
Iteration 2900: with minibatch training loss = 0.183 and accuracy of 0.34
Iteration 3000: with minibatch training loss = 0.187 and accuracy of 0.42
Epoch 4, Overall loss = 1.31 and accuracy of 0.498
Iteration 3100: with minibatch training loss = 0.173 and accuracy of 0.45
Iteration 3200: with minibatch training loss = 0.134 and accuracy of 0.64
Iteration 3300: with minibatch training loss = 0.155 and accuracy of 0.47
Iteration 3400: with minibatch training loss = 0.141 and accuracy of 0.61
Iteration 3500: with minibatch training loss = 0.147 and accuracy of 0.53
Iteration 3600: with minibatch training loss = 0.141 and accuracy of 0.59
Iteration 3700: with minibatch training loss = 0.158 and accuracy of 0.56
Iteration 3800: with minibatch training loss = 0.163 and accuracy of 0.48
Epoch 5, Overall loss = 1.46 and accuracy of 0.533
Iteration 3900: with minibatch training loss = 0.134 and accuracy of 0.56
Iteration 4000: with minibatch training loss = 0.141 and accuracy of 0.66
Iteration 4100: with minibatch training loss = 0.136 and accuracy of 0.7
Iteration 4200: with minibatch training loss = 0.117 and accuracy of 0.73
Iteration 4300: with minibatch training loss = 0.118 and accuracy of 0.7
Iteration 4400: with minibatch training loss = 0.156 and accuracy of 0.52
Iteration 4500: with minibatch training loss = 0.143 and accuracy of 0.48
Epoch 6, Overall loss = 1.6 and accuracy of 0.586
Iteration 4600: with minibatch training loss = 0.143 and accuracy of 0.61
Iteration 4700: with minibatch training loss = 0.116 and accuracy of 0.62
Iteration 4800: with minibatch training loss = 0.11 and accuracy of 0.66
Iteration 4900: with minibatch training loss = 0.124 and accuracy of 0.67
Iteration 5000: with minibatch training loss = 0.102 and accuracy of 0.73
Iteration 5100: with minibatch training loss = 0.138 and accuracy of 0.64
Iteration 5200: with minibatch training loss = 0.118 and accuracy of 0.75
Iteration 5300: with minibatch training loss = 0.14 and accuracy of 0.58
Epoch 7, Overall loss = 1.73 and accuracy of 0.645
Iteration 5400: with minibatch training loss = 0.125 and accuracy of 0.64
Iteration 5500: with minibatch training loss = 0.105 and accuracy of 0.75
Iteration 5600: with minibatch training loss = 0.127 and accuracy of 0.67
Iteration 5700: with minibatch training loss = 0.132 and accuracy of 0.67
Iteration 5800: with minibatch training loss = 0.113 and accuracy of 0.66
Iteration 5900: with minibatch training loss = 0.131 and accuracy of 0.66
Iteration 6000: with minibatch training loss = 0.101 and accuracy of 0.72
Iteration 6100: with minibatch training loss = 0.0987 and accuracy of 0.69
Epoch 8, Overall loss = 1.85 and accuracy of 0.678
Iteration 6200: with minibatch training loss = 0.114 and accuracy of 0.7
Iteration 6300: with minibatch training loss = 0.0803 and accuracy of 0.81
Iteration 6400: with minibatch training loss = 0.14 and accuracy of 0.58
Iteration 6500: with minibatch training loss = 0.0977 and accuracy of 0.78
Iteration 6600: with minibatch training loss = 0.12 and accuracy of 0.69
Iteration 6700: with minibatch training loss = 0.129 and accuracy of 0.66
Iteration 6800: with minibatch training loss = 0.107 and accuracy of 0.67
Epoch 9, Overall loss = 1.96 and accuracy of 0.708
Iteration 6900: with minibatch training loss = 0.108 and accuracy of 0.77
Iteration 7000: with minibatch training loss = 0.0929 and accuracy of 0.75
Iteration 7100: with minibatch training loss = 0.118 and accuracy of 0.69
Iteration 7200: with minibatch training loss = 0.114 and accuracy of 0.69
Iteration 7300: with minibatch training loss = 0.0893 and accuracy of 0.77
Iteration 7400: with minibatch training loss = 0.086 and accuracy of 0.77
Iteration 7500: with minibatch training loss = 0.0924 and accuracy of 0.75
Iteration 7600: with minibatch training loss = 0.0976 and accuracy of 0.72
Epoch 10, Overall loss = 2.06 and accuracy of 0.733
```



```
Validation
Epoch 1, Overall loss = 0.149 and accuracy of 0.624
```

```
Out[34]: (0.1494178763628006,
          0.624,
          [7.130663871765137,
           10.43324089050293,
           7.5779852867126465,
           8.036458969116211,
           10.860355377197266,
           8.147161483764648,
           10.744089126586914,
           9.59776782989502,
           10.225924491882324,
           8.839098930358887,
           9.92397689819336,
           9.994573593139648,
           12.471409797668457,
           8.962006568908691,
           9.86272144317627,
           6.61044180393219])
```

```
In [35]: # Test your model here, and make sure
         # the output of this cell is the accuracy
```

```
        # of your best model on the training and val sets
        # We're looking for >= 60% accuracy on Validation
        print('Training')
        run_model(sess,y_out,mean_loss,X_train,y_train,1,64)
        print('Validation')
        run_model(sess,y_out,mean_loss,X_val,y_val,1,64)
```

```
Training
Epoch 1, Overall loss = 0.104 and accuracy of 0.735
Validation
Epoch 1, Overall loss = 0.148 and accuracy of 0.622


Out[35]: (0.14842549794912338,
          0.622,
          [10.160228729248047,
           7.021052360534668,
           9.489645004272461,
           9.791773796081543,
           10.264333724975586,
           7.299481391906738,
           8.497721672058105,
           10.746976852416992,
           9.730812072753906,
           8.981893539428711,
           9.984275817871094,
           11.433265686035156,
           9.882766723632812,
           10.927777290344238,
           9.342181205749512,
           4.871312081813812])
```

1.2.5 Briefly describe what you did here

In this cell you should also write an explanation of what you did, any additional features that you implemented, and any visualizations or graphs that you make in the process of training and evaluating your network

With reference to https://github.com/adhishthite/cifar10-optimizers, ADAM optimzer seems to be the best. Thus ADAM is used.

The model is an attempt to replicate the VGG model which comprises of a bunch of features (varies depending on vgg version. This is vgg13). These features contain double conv2d layers to a maxpool layer. The model is reduced (removed the last 4 conv2d layer) to decrease train time and complexity of implementation. The features are then followed up with an adaptive pool layer, followed by their classifier layer which has Linear -> ReLU -> Dropout -> Linear -> ReLU -> Dropout -> Linear

More training could have possibly improved the accuracy. However, the computation time was very long (10 epochs in about 1 and a half hour)

1.2.6 Test Set - DO THIS ONLY ONCE

Now that we've gotten a result that we're happy with, we test our final model on the test set. This would be the score we would achieve on a competition. Think about how this compares to your validation set accuracy.

```
In [36]: print('Test')
         run_model(sess,y_out,mean_loss,X_test,y_test,1,64)
```

```
Test
Epoch 1, Overall loss = 0.154 and accuracy of 0.627
```

```
Out[36]: (0.15379231536388396,
          0.6273,
          [11.33283519744873,
           9.795106887817383,
           9.47136402130127,
           9.999404907226562,
           10.129778861999512,
           10.624198913574219,
           8.981740951538086,
           9.515828132629395,
           9.191923141479492,
           8.909574508666992,
           10.234477043151855,
           8.994105339050293,
           10.58629035949707,
           8.258219718933105,
           9.180534362792969,
           11.137691497802734,
           10.507099151611328,
           11.750336647033691,
           7.494290828704834,
           9.29546070098877,
           13.096395492553711,
           8.93061351776123,
           8.560534477233887,
           9.374885559082031,
           11.801852226257324,
           10.040114402770996,
           7.055781364440918,
           11.515698432922363,
           8.728960037231445,
           9.457276344299316,
           8.279126167297363,
           10.40467643737793,
           9.693547248840332,
           7.84722375869751,
           9.800959587097168,
           10.03348159790039,
           10.707991600036621,
           9.473005294799805,
           8.182374954223633,
           9.665445327758789,
           10.56529426574707,
           10.616511344909668,
           9.069645881652832,
           11.619061470031738,
           9.11230182647705,
           9.940861701965332,
           10.283823013305664,
           13.429156303405762,
           9.14712905883789,
           9.278017044067383,
           7.708006858825684,
           10.207348823547363,
           10.802620887756348,
           12.643410682678223,
           8.50526237487793,
           8.894704818725586,
           8.155600547790527,
           6.899369716644287,
           9.691670417785645,
           10.39765453338623,
           9.244952201843262,
           11.503959655761719,
           8.716822624206543,
           14.38934326171875,
           10.087177276611328,
           10.360298156738281,
           8.965336799621582,
           9.8948974609375,
           10.973193168640137,
           9.240870475769043,
           8.504170417785645,
           9.867586135864258,
           10.71095085144043,
           9.690818786621094,
           8.6669282913208,
           9.653347969055176,
           9.427586555480957,
           8.80008602142334,
           8.30834674835205,
           7.881944179534912,
           9.44032096862793,
           8.098142623901367,
           11.30392074584961,
           9.187832832336426,
           8.88115119934082,
           11.105608940124512,
           11.144731521606445,
           10.347967147827148,
           9.80964469909668,
           9.097892761230469,
           12.250035285949707,
           8.387471199035645,
           12.39819049835205,
           8.26802921295166,
           10.239764213562012,
           11.009922981262207,
           11.552200317382812,
           10.591276168823242,
           10.043190002441406,
           10.973795890808105,
           10.547910690307617,
           11.965154647827148,
           9.0882568359375,
           9.242231369018555,
           10.2449951171875,
           11.540396690368652,
           12.228995323181152,
           9.744340896606445,
           12.237414360046387,
           8.485617637634277,
           7.903608798980713,
           10.330392837524414,
           11.00825023651123,
           9.54046630859375,
           12.17712116241455,
           9.993121147155762,
           9.819668769836426,
           10.624098777770996,
           10.23338508605957,
           8.99186897277832,
           13.5558443069458,
           6.834461212158203,
           7.603018283843994,
           8.450637817382812,
           9.64976692199707,
           8.950708389282227,
           8.929677963256836,
           7.8212890625,
           11.287590026855469,
           11.054445266723633,
           9.574865341186523,
           11.93579387664795,
           9.459417343139648,
           9.604421615600586,
           10.116786003112793,
           9.83716869354248,
           8.224587440490723,
           7.0352630615234375,
           11.4835844039917,
           10.904444694519043,
           12.015095710754395,
           9.762120246887207,
           8.231534004211426,
           9.998664855957031,
           10.347798347473145,
           9.316227912902832,
           9.68592643737793,
           7.331617832183838,
           6.9308366775512695,
           8.157957077026367,
           8.365205764770508,
           10.082704544067383,
           11.496845245361328,
           11.869211196899414,
           9.182756423950195,
           10.205581665039062,
           2.654639482498169])
```