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.

Acknowledgement: This exercise is adapted from Stanford CS231n.

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 2.0. Most examples on the web today are still in 1.x, so be careful not to confuse the two when looking up documentation.

Install Tensorflow 2.0

Tensorflow 2.0 is still not in a fully 100% stable release, but it's still usable and more intuitive than TF 1.x. Please make sure you have it installed before moving on in this notebook! Here are some steps to get started:

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/pip

A guide on creating Anaconda enviornments: https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/20/conda/

This will give you an new enviornemnt to play in TF 2.0. Generally, if you plan to also use TensorFlow in your other projects, you might also want to keep a seperate Conda environment or virtualenv in Python 3.7 that has Tensorflow 1.9, so you can switch back and forth at will.

Acknowledgement: This exercise is adapted from Stanford CS231n.

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.

Part I: Preparation

In [21]:

```
import os
import tensorflow as tf
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt

%matplotlib inline
```

In [22]:

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

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

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

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

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

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

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [23]:

```
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)
```

In [24]:

```
# 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**. It's not neccessary to use a GPU for this assignment; if you are working on Google Cloud then we recommend that you do not use a GPU, as it will be significantly more expensive.

In [25]:

```
# Set up some global variables
USE_GPU = True

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

# Constant to control how often we print when training models
print_every = 100

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

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

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.

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.

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.

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.

In [26]:

```
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))
```

In [27]:

```
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)
```

Barebones TensorFlow: Define a Two-Layer Network

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

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

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

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

In [28]:

```
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
```

In [29]:

```
def two_layer_fc_test():
    hidden_layer_size = 42

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

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

    print(scores.shape)

two_layer_fc_test()
```

```
(64, 10)
```

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

In [30]:

```
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)*****

    padding1 = [[0,0], [2,2], [2,2], [0,0]]
    conv1 = tf.nn.relu(tf.nn.conv2d(x, conv_w1, 1, padding1) + conv_b1)

    padding2 = [[0,0], [1,1], [1,1], [0,0]]
    conv2 = tf.nn.relu(tf.nn.conv2d(conv1, conv_w2, 1, padding2) + conv_b2)

    scores = tf.matmul(flatten(conv2), 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, 10)`.

In [31]:

```
def three_layer_convnet_test():

    with tf.device(device):
        x = tf.zeros((64, 32, 32, 3))
        conv_w1 = tf.zeros((5, 5, 3, 6))
        conv_b1 = tf.zeros((6,))
        conv_w2 = tf.zeros((3, 3, 6, 9))
        conv_b2 = tf.zeros((9,))
        fc_w = tf.zeros((32 * 32 * 9, 10))
        fc_b = tf.zeros((10,))
        params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
        scores = three_layer_convnet(x, params)

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

three_layer_convnet_test()
```

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

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

In [32]:

```python
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
```

In [33]:

```python
def train_part2(model_fn, init_fn, learning_rate, epochs):
    """
    Train a model on CIFAR-10.

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


    params = init_fn()  # Initialize the model parameters
    for e in range(epochs):
        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('Epoch %d, iteration %d, loss = %.4f' % (e, t, loss))
                print('Validation:')
                check_accuracy(val_dset, model_fn, params)
    return params
```

In [34]:

```python
def check_accuracy(dset, 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))
```

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

In [35]:

```python
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)
```

Barebones TensorFlow: Train a Two-Layer Network

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

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

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

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

In [36]:

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

    Inputs: None

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

learning_rate = 1e-2
print('Train')
trained_params = train_part2(two_layer_fc, two_layer_fc_init, learning_rate,5)
print('Done!')
```

```
Train
Epoch 0, iteration 0, loss = 3.1704
Validation:
     Got 134 / 1000 correct (13.40%)
Epoch 0, iteration 100, loss = 1.8649
Validation:
     Got 364 / 1000 correct (36.40%)
Epoch 0, iteration 200, loss = 1.4961
Validation:
     Got 370 / 1000 correct (37.00%)
Epoch 0, iteration 300, loss = 1.7581
Validation:
     Got 373 / 1000 correct (37.30%)
Epoch 0, iteration 400, loss = 1.7378
Validation:
     Got 420 / 1000 correct (42.00%)
Epoch 0, iteration 500, loss = 1.8353
Validation:
     Got 429 / 1000 correct (42.90%)
Epoch 0, iteration 600, loss = 1.8438
Validation:
     Got 421 / 1000 correct (42.10%)
Epoch 0, iteration 700, loss = 1.9588
Validation:
     Got 444 / 1000 correct (44.40%)
Epoch 1, iteration 0, loss = 1.4847
Validation:
     Got 438 / 1000 correct (43.80%)
Epoch 1, iteration 100, loss = 1.5221
Validation:
     Got 486 / 1000 correct (48.60%)
Epoch 1, iteration 200, loss = 1.2549
Validation:
     Got 457 / 1000 correct (45.70%)
Epoch 1, iteration 300, loss = 1.5399
Validation:
     Got 454 / 1000 correct (45.40%)
Epoch 1, iteration 400, loss = 1.4310
Validation:
     Got 468 / 1000 correct (46.80%)
Epoch 1, iteration 500, loss = 1.6177
Validation:
     Got 462 / 1000 correct (46.20%)
Epoch 1, iteration 600, loss = 1.6477
Validation:
     Got 459 / 1000 correct (45.90%)
Epoch 1, iteration 700, loss = 1.6875
Validation:
     Got 489 / 1000 correct (48.90%)
Epoch 2, iteration 0, loss = 1.3044
Validation:
     Got 469 / 1000 correct (46.90%)
Epoch 2, iteration 100, loss = 1.4160
Validation:
     Got 503 / 1000 correct (50.30%)
Epoch 2, iteration 200, loss = 1.1234
Validation:
     Got 482 / 1000 correct (48.20%)
Epoch 2, iteration 300, loss = 1.4046
Validation:
     Got 474 / 1000 correct (47.40%)
Epoch 2, iteration 400, loss = 1.2766
Validation:
     Got 483 / 1000 correct (48.30%)
Epoch 2, iteration 500, loss = 1.4995
Validation:
     Got 482 / 1000 correct (48.20%)
Epoch 2, iteration 600, loss = 1.4968
Validation:
     Got 478 / 1000 correct (47.80%)
Epoch 2, iteration 700, loss = 1.5458
Validation:
     Got 499 / 1000 correct (49.90%)
Epoch 3, iteration 0, loss = 1.1924
Validation:
     Got 488 / 1000 correct (48.80%)
Epoch 3, iteration 100, loss = 1.3300
Validation:
     Got 520 / 1000 correct (52.00%)
Epoch 3, iteration 200, loss = 1.0280
Validation:
     Got 500 / 1000 correct (50.00%)
Epoch 3, iteration 300, loss = 1.2997
Validation:
     Got 489 / 1000 correct (48.90%)
Epoch 3, iteration 400, loss = 1.1613
Validation:
     Got 490 / 1000 correct (49.00%)
Epoch 3, iteration 500, loss = 1.4032
Validation:
     Got 497 / 1000 correct (49.70%)
Epoch 3, iteration 600, loss = 1.3779
Validation:
     Got 489 / 1000 correct (48.90%)
Epoch 3, iteration 700, loss = 1.4445
Validation:
     Got 509 / 1000 correct (50.90%)
Epoch 4, iteration 0, loss = 1.0922
Validation:
     Got 506 / 1000 correct (50.60%)
Epoch 4, iteration 100, loss = 1.2457
Validation:
     Got 522 / 1000 correct (52.20%)
Epoch 4, iteration 200, loss = 0.9499
Validation:
```

```
     Got 509 / 1000 correct (50.90%)
Epoch 4, iteration 300, loss = 1.2209
Validation:
     Got 502 / 1000 correct (50.20%)
Epoch 4, iteration 400, loss = 1.0616
Validation:
     Got 498 / 1000 correct (49.80%)
Epoch 4, iteration 500, loss = 1.3121
Validation:
     Got 502 / 1000 correct (50.20%)
Epoch 4, iteration 600, loss = 1.2766
Validation:
     Got 485 / 1000 correct (48.50%)
Epoch 4, iteration 700, loss = 1.3592
Validation:
     Got 517 / 1000 correct (51.70%)
Done!
```

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 [37]:

```
print('Test')
check_accuracy(test_dset, two_layer_fc, trained_params)
```

```
Test
     Got 5036 / 10000 correct (50.36%)
```

Barebones TensorFlow: Train a three-layer ConvNet

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

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

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

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

In [38]:

```
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(np.zeros([32]),dtype=tf.float32)
    conv_w2 = tf.Variable(create_matrix_with_kaiming_normal((3,3,32,16)))
    conv_b2 = tf.Variable(np.zeros([16]),dtype=tf.float32)
    fc_w = tf.Variable(create_matrix_with_kaiming_normal((32*32*16,10)))
    fc_b = tf.Variable(np.zeros([10]),dtype=tf.float32)

    params = [conv_w1,conv_b1,conv_w2,conv_b2,fc_w,fc_b]

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

learning_rate = 3e-3
train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate,5)
```

```
Epoch 0, iteration 0, loss = 2.7989
Validation:
     Got 106 / 1000 correct (10.60%)
Epoch 0, iteration 100, loss = 1.9118
Validation:
     Got 350 / 1000 correct (35.00%)
Epoch 0, iteration 200, loss = 1.5325
Validation:
     Got 389 / 1000 correct (38.90%)
Epoch 0, iteration 300, loss = 1.8759
Validation:
     Got 384 / 1000 correct (38.40%)
Epoch 0, iteration 400, loss = 1.7855
Validation:
     Got 423 / 1000 correct (42.30%)
Epoch 0, iteration 500, loss = 1.6878
Validation:
     Got 427 / 1000 correct (42.70%)
Epoch 0, iteration 600, loss = 1.7568
Validation:
     Got 454 / 1000 correct (45.40%)
Epoch 0, iteration 700, loss = 1.6619
Validation:
     Got 424 / 1000 correct (42.40%)
Epoch 1, iteration 0, loss = 1.4683
Validation:
     Got 454 / 1000 correct (45.40%)
Epoch 1, iteration 100, loss = 1.4329
Validation:
     Got 454 / 1000 correct (45.40%)
Epoch 1, iteration 200, loss = 1.2439
Validation:
     Got 483 / 1000 correct (48.30%)
Epoch 1, iteration 300, loss = 1.6175
Validation:
     Got 472 / 1000 correct (47.20%)
Epoch 1, iteration 400, loss = 1.4671
Validation:
     Got 482 / 1000 correct (48.20%)
Epoch 1, iteration 500, loss = 1.5604
Validation:
     Got 482 / 1000 correct (48.20%)
Epoch 1, iteration 600, loss = 1.6193
Validation:
     Got 505 / 1000 correct (50.50%)
Epoch 1, iteration 700, loss = 1.5099
Validation:
     Got 486 / 1000 correct (48.60%)
Epoch 2, iteration 0, loss = 1.2974
Validation:
     Got 510 / 1000 correct (51.00%)
Epoch 2, iteration 100, loss = 1.3211
Validation:
     Got 521 / 1000 correct (52.10%)
Epoch 2, iteration 200, loss = 1.1137
Validation:
     Got 512 / 1000 correct (51.20%)
Epoch 2, iteration 300, loss = 1.4767
Validation:
     Got 504 / 1000 correct (50.40%)
Epoch 2, iteration 400, loss = 1.2762
Validation:
     Got 520 / 1000 correct (52.00%)
Epoch 2, iteration 500, loss = 1.4586
Validation:
     Got 524 / 1000 correct (52.40%)
Epoch 2, iteration 600, loss = 1.5406
Validation:
     Got 530 / 1000 correct (53.00%)
Epoch 2, iteration 700, loss = 1.4227
Validation:
     Got 519 / 1000 correct (51.90%)
Epoch 3, iteration 0, loss = 1.1941
Validation:
     Got 544 / 1000 correct (54.40%)
Epoch 3, iteration 100, loss = 1.2533
Validation:
     Got 544 / 1000 correct (54.40%)
Epoch 3, iteration 200, loss = 1.0270
Validation:
     Got 534 / 1000 correct (53.40%)
Epoch 3, iteration 300, loss = 1.3866
Validation:
     Got 520 / 1000 correct (52.00%)
Epoch 3, iteration 400, loss = 1.1716
Validation:
     Got 537 / 1000 correct (53.70%)
Epoch 3, iteration 500, loss = 1.3840
Validation:
     Got 532 / 1000 correct (53.20%)
Epoch 3, iteration 600, loss = 1.4903
Validation:
     Got 547 / 1000 correct (54.70%)
Epoch 3, iteration 700, loss = 1.3626
Validation:
     Got 543 / 1000 correct (54.30%)
Epoch 4, iteration 0, loss = 1.1161
Validation:
     Got 546 / 1000 correct (54.60%)
Epoch 4, iteration 100, loss = 1.1930
Validation:
     Got 556 / 1000 correct (55.60%)
Epoch 4, iteration 200, loss = 0.9620
Validation:
     Got 545 / 1000 correct (54.50%)
Epoch 4, iteration 300, loss = 1.3041
Validation:
     Got 532 / 1000 correct (53.20%)
Epoch 4, iteration 400, loss = 1.0983
Validation:
     Got 553 / 1000 correct (55.30%)
Epoch 4, iteration 500, loss = 1.3185
Validation:
     Got 547 / 1000 correct (54.70%)
Epoch 4, iteration 600, loss = 1.4434
Validation:
     Got 561 / 1000 correct (56.10%)
Epoch 4, iteration 700, loss = 1.3026
Validation:
     Got 556 / 1000 correct (55.60%)
```

Out[38]:

```
[<tf.Variable 'Variable:0' shape=(5, 5, 3, 32) dtype=float32, numpy=
 array([[[[ 5.56443993e-04,  1.18631206e-01,  6.97398037e-02, ...,
           -1.84053168e-01, -1.66528840e-02,  9.97290984e-02],
          [-3.45993817e-01, -1.50569111e-01, -7.71120936e-02, ...,
            9.76547897e-02, -1.52591571e-01, -2.65006363e-01],
          [ 1.98487565e-02,  1.20968789e-01,  2.08010316e-01, ...,
            1.55505940e-01, -6.83139041e-02, -8.37179050e-02]],

         [[-1.11844465e-01,  9.13498551e-02, -3.41508314e-02, ...,
            1.90325066e-01,  2.52548397e-01,  4.90393825e-02],
          [-2.60239579e-02, -8.85520130e-02, -2.65162051e-01, ...,
            8.80659968e-02,  1.62238926e-01, -3.27277146e-02],
          [-1.43725619e-01, -1.16614625e-01,  1.19238771e-01, ...,
            1.79389417e-01,  1.95420265e-01, -6.74604177e-02]],

         [[-2.52454691e-02, -1.87383026e-01,  5.11932261e-02, ...,
           -2.58032709e-01,  1.64115265e-01, -8.00734907e-02],
          [-4.11269395e-03, -2.50206113e-01,  1.63126096e-01, ...,
            1.02143869e-01,  6.65559471e-02, -1.00496113e-01],
          [ 1.26681596e-01, -2.41233796e-01, -1.78175524e-01, ...,
           -4.18260135e-02, -9.75468382e-02, -1.17716752e-01]],

         [[ 1.10411784e-02,  1.56573176e-01, -6.49241135e-02, ...,
           -2.70640068e-02, -1.17770940e-01, -9.75733548e-02],
          [-1.67433202e-01,  2.03393072e-01,  2.34874085e-01, ...,
            1.72961563e-01, -6.01877347e-02, -3.98627594e-02],
          [-3.98935117e-02,  2.63539582e-01,  1.49242133e-01, ...,
            3.80536348e-01,  1.01395976e-03,  2.62944754e-02]],

         [[ 2.90422589e-02, -1.29476920e-01,  1.07850105e-01, ...,
           -6.86842995e-03, -9.96568948e-02,  1.73874989e-01],
          [ 1.01059787e-01,  2.90403068e-01,  1.04186244e-01, ...,
            1.13830097e-01, -5.54877967e-02,  2.17784703e-01],
          [-2.59238362e-01, -4.00623754e-02, -1.55309677e-01, ...,
           -1.64319530e-01,  2.24734824e-02, -2.94008348e-02]]],


        [[[-2.86124438e-01,  1.82434455e-01, -2.06895038e-01, ...,
           -1.49360776e-01, -3.49962384e-01,  1.36372000e-01],
          [-3.08762658e-02, -3.46992344e-01,  1.08081892e-01, ...,
            4.62648235e-02,  3.82333584e-02, -5.82093775e-01],
          [-1.30524859e-01, -9.56613868e-02, -8.48294944e-02, ...,
            1.60843760e-01, -9.31482166e-02,  3.81414290e-03]],

         [[-9.87570509e-02,  1.12265006e-01,  9.53418538e-02, ...,
           -3.17377634e-02, -9.00995210e-02,  2.21553668e-02],
          [-5.15227616e-02, -1.11026019e-01,  1.07829206e-01, ...,
            4.57606325e-03, -1.23800643e-01,  5.40949926e-02],
          [-2.01739427e-02,  1.67001843e-01, -1.11211419e-01, ...,
           -8.27369615e-02,  3.42498064e-01,  6.53060228e-02]],

         [[-1.96375959e-02,  1.78206563e-01,  5.36187291e-02, ...,
           -7.84516893e-03,  2.34627239e-02,  3.20617467e-01],
          [ 2.03259662e-02,  2.32067868e-01, -1.85930148e-01, ...,
           -2.26653844e-01,  7.05135763e-02, -9.15407985e-02],
          [ 4.83352430e-02, -1.77630335e-02,  7.76318237e-02, ...,
           -1.10168837e-01, -1.10451616e-01, -3.03407442e-02]],

         [[-1.65438190e-01, -2.88932454e-02, -5.50457723e-02, ...,
            1.89462781e-01, -2.45819539e-01, -1.72260568e-01],
          [-1.51417404e-02,  1.46931902e-01,  3.76393139e-01, ...,
           -1.57464713e-01, -1.14196874e-01,  5.15166968e-02],
          [-7.45943636e-02, -1.98911026e-01, -2.42571741e-01, ...,
           -1.79949209e-01, -7.71821141e-02, -1.92603454e-01]],

         [[-1.10606186e-01,  2.49668434e-01,  1.57272786e-01, ...,
           -4.28163260e-03, -1.52902842e-01, -1.46827670e-02],
          [-1.05624899e-01, -2.13669568e-01, -7.63649270e-02, ...,
           -1.41421944e-01, -2.72439361e-01, -2.18311980e-01],
          [-3.62758115e-02,  9.84072536e-02, -5.92771433e-02, ...,
            1.23992711e-01,  9.25617218e-02,  1.15382150e-01]]],


        [[[ 1.23171518e-02, -3.16217780e-01, -1.00884978e-02, ...,
            5.37856333e-02, -2.22149178e-01,  9.99258459e-03],
          [-1.55091390e-01, -1.38913125e-01,  3.14529762e-02, ...,
           -1.03334740e-01,  2.08016932e-02, -1.82168454e-01],
          [ 2.30989501e-01, -1.78890660e-01, -1.09276973e-01, ...,
            1.41699761e-02,  6.63794801e-02,  1.30568236e-01]],

         [[ 8.42168257e-02,  2.87086189e-01, -1.11402854e-01, ...,
            4.52377982e-02, -1.46608865e-02,  3.03827345e-01],
          [ 3.63097668e-01,  2.02704608e-01, -3.00714731e-01, ...,
           -2.97589839e-01, -5.22660434e-01, -1.69486746e-01],
          [ 2.74245292e-01, -7.36232549e-02, -1.68348193e-01, ...,
            1.59809068e-02, -1.38680801e-01,  2.36625642e-01]],

         [[-2.13166833e-01, -8.49105567e-02,  1.03632398e-01, ...,
           -6.53526112e-02,  1.08042084e-01, -8.00037533e-02],
          [-1.86927602e-01,  2.02168077e-01,  8.26876312e-02, ...,
           -8.85963906e-04, -1.71832383e-01,  9.28165913e-02],
          [-6.47939518e-02, -6.92095831e-02,  3.36634696e-01, ...,
            4.77803163e-02, -1.01414695e-01,  1.35543063e-01]],

         [[-1.37224346e-01, -2.03122571e-01, -2.95330495e-01, ...,
           -1.18960075e-01,  1.20186120e-01, -8.44164267e-02],
          [-5.25961854e-02, -2.63099018e-02,  1.96577877e-01, ...,
           -1.43589094e-01,  3.45242739e-01, -6.70529380e-02],
          [ 1.25749782e-01,  5.43028750e-02, -2.42148593e-01, ...,
           -5.22983819e-02,  1.11512788e-01, -7.28580775e-03]],

         [[ 7.18552526e-03,  4.63072117e-03,  1.99692965e-01, ...,
           -5.60920034e-03,  5.27970381e-02, -2.13622232e-03],
          [ 1.42661422e-01,  1.93364337e-01, -1.70606837e-01, ...,
            7.05385953e-03, -1.23900793e-01,  9.59951133e-02],
          [-3.81700397e-01,  1.30459880e-02, -1.41793638e-01, ...,
           -1.81762651e-01, -3.94907147e-02, -1.89669952e-01]]],


        [[[ 2.99702883e-02,  5.86231314e-02, -1.25816375e-01, ...,
            4.80809808e-02,  6.38354793e-02,  8.41939449e-02],
          [ 1.49857819e-01, -1.35603473e-01,  2.46832911e-02, ...,
           -2.97899634e-01, -1.02248564e-01, -9.85182375e-02],
          [ 2.16291398e-01,  1.65728316e-01, -1.75435860e-02, ...,
            7.83121213e-02,  8.31533149e-02,  2.16550052e-01]],

         [[ 1.90119386e-01,  1.78421929e-01, -1.08114786e-01, ...,
            3.08209270e-01,  4.38746251e-02,  8.62762928e-02],
          [ 1.83774129e-01,  1.16906710e-01,  2.26200670e-01, ...,
           -8.38240385e-02, -9.37787741e-02, -2.86806494e-01],
          [ 5.67776524e-02, -8.88782740e-02,  2.16801986e-01, ...,
           -8.82271305e-02, -1.64053589e-01,  6.68115448e-03]],

         [[-1.63530037e-01,  1.24297574e-01, -4.25106347e-01, ...,
           -2.00975668e-02,  6.84082359e-02, -7.51050636e-02],
          [-3.38331498e-02, -4.27612029e-02,  9.48979631e-02, ...,
            6.60735816e-02, -7.16675669e-02,  1.80997282e-01],
          [ 2.36374766e-01, -2.88259313e-02, -3.05780005e-02, ...,
           -1.89803496e-01,  9.76248458e-03, -8.34640414e-02]],

         [[-1.19703151e-01, -6.32570311e-02,  1.68777168e-01, ...,
            4.04719673e-02,  2.53647000e-01, -1.25679914e-02],
          [ 2.66644746e-01,  3.15182731e-02,  6.63963705e-02, ...,
           -2.12355927e-02, -1.09362625e-01,  4.11595702e-02],
          [-2.54486531e-01,  1.55811787e-01, -3.54972094e-01, ...,
            1.14905380e-01, -1.14335753e-01,  2.24685818e-01]],

         [[ 1.94291726e-01,  2.05846742e-01, -2.45133955e-02, ...,
           -7.09777325e-02,  1.73036382e-01,  4.37633842e-02],
          [ 1.46225542e-01, -1.28179751e-02, -1.61483571e-01, ...,
           -9.00074467e-03,  8.71978700e-02, -3.07628751e-01],
          [-1.62551939e-01,  1.85657695e-01, -9.42396373e-02, ...,
            3.45536143e-01, -1.41372457e-01, -1.21281072e-01]]],


        [[[ 1.70321167e-01, -2.77139068e-01, -2.84414738e-01, ...,
            2.71182895e-01,  5.57033122e-02,  1.26124963e-01],
          [ 9.85246971e-02, -1.57869294e-01, -6.75132275e-02, ...,
            1.21933194e-02, -2.56901890e-01, -2.81368941e-01],
          [ 4.99382475e-03, -2.01647028e-01,  6.91522062e-02, ...,
           -3.30034681e-02, -5.39601296e-02,  5.50635941e-02]],

         [[-8.75671655e-02,  3.53732929e-02,  2.34812740e-02, ...,
           -1.93153601e-02,  8.08140188e-02, -6.10946976e-02],
          [ 1.37669906e-01, -5.53138293e-02,  3.38312238e-01, ...,
            2.74507493e-01,  1.24993257e-01, -2.93875672e-02],
          [-1.41938254e-01,  3.35920267e-02, -1.23093046e-01, ...,
            3.29923928e-02, -5.70078343e-02, -1.58600494e-01]],

         [[ 9.66815725e-02, -1.16262930e-02,  2.42204770e-01, ...,
            1.11329868e-01, -5.63795902e-02, -1.03267550e-01],
          [-6.55370653e-02,  2.58234330e-02, -9.94272158e-02, ...,
            2.05987748e-02,  3.44549790e-02, -7.26835057e-02],
          [-1.16415918e-02, -4.12603244e-02,  2.85417289e-01, ...,
           -4.97911200e-02, -7.60417953e-02,  3.97710055e-02]],

         [[-1.73065979e-02, -9.94748101e-02,  1.11887410e-01, ...,
           -2.16378737e-02,  7.63203055e-02, -1.86387926e-01],
          [ 5.63118346e-02, -1.37527898e-01, -1.53100446e-01, ...,
           -6.34386688e-02, -6.64559901e-02,  2.64848530e-01],
          [ 5.60273649e-03,  4.20562625e-01, -2.69303352e-01, ...,
           -2.74846703e-01,  1.74455926e-01, -1.48143679e-01]],

         [[-2.04778295e-02,  1.64461508e-01,  3.20098887e-04, ...,
            2.01444566e-01, -5.68536445e-02,  1.17680460e-01],
          [ 3.12834114e-01,  1.15634063e-02, -1.84514970e-02, ...,
           -1.04381979e-01, -5.92245907e-03, -5.63395768e-02],
          [ 1.23652533e-01,  8.73611793e-02, -1.11411273e-01, ...,
            1.21632926e-01, -1.07444949e-01, -1.29844531e-01]]]],
       dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(32,) dtype=float32, numpy=
 array([-0.06075452,  0.06717905, -0.02265327,  0.00792555, -0.00986119,
         0.11976448,  0.01113429,  0.119619  , -0.01664783,  0.04873193,
        -0.0853018 ,  0.01282979,  0.00839457, -0.08521993,  0.00345913,
        -0.02686683,  0.04585355,  0.03206836,  0.01718122, -0.00353727,
        -0.01105129,  0.02080524, -0.01293921, -0.04477442, -0.07059418,
         0.03745364,  0.02010462, -0.00935231, -0.04295825,  0.00273019,
        -0.01051593, -0.00707307], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(3, 3, 32, 16) dtype=float32, numpy=
 array([[[[-8.21002573e-02, -1.56082427e-02, -9.46564525e-02, ...,
            8.89036357e-02, -6.19673803e-02,  3.67258191e-02],
          [-1.32351503e-01, -2.18915921e-02,  4.15199809e-02, ...,
            6.21181093e-02, -1.23336865e-03, -1.51720475e-02],
          [-1.21895084e-02,  7.32530132e-02, -6.39437735e-02, ...,
           -1.35172578e-02, -1.61584049e-01,  3.61367986e-02],
          ...,
          [ 9.17640179e-02, -2.20617950e-02,  1.56560481e-01, ...,
           -1.03838526e-01, -1.46196485e-01, -4.92461249e-02],
          [-5.97426966e-02,  1.33313984e-01, -1.73651576e-02, ...,
           -2.58815009e-02, -1.53525323e-01, -1.11026101e-01],
          [-3.35389823e-02, -2.11681232e-01,  8.98545161e-02, ...,
            5.36768464e-03, -4.82794903e-02, -3.06677017e-02]],

         [[-3.54224592e-02,  1.78814247e-01, -7.24029094e-02, ...,
           -6.09257407e-02,  3.95964421e-02,  9.50392522e-03],
          [ 3.25119872e-05, -7.42026232e-03,  5.46423532e-02, ...,
            1.23963542e-01, -1.28053352e-01,  2.68960409e-02],
          [-7.37031177e-02, -1.12398461e-01, -5.68695255e-02, ...,
           -6.71190396e-02,  2.31024660e-02,  6.48080558e-02],
          ...,
          [ 5.25600277e-03, -6.78709522e-02, -1.21569514e-01, ...,
            3.66782025e-02, -3.06745954e-02, -1.12102769e-01],
          [-3.17764319e-02,  3.54128592e-02,  9.89639312e-02, ...,
            8.00145194e-02,  1.25448674e-01,  4.40859683e-02],
          [-1.94795076e-02,  4.41553928e-02, -2.98654102e-02, ...,
            7.05937967e-02, -2.19624434e-02, -9.16193426e-02]],

         [[-1.16664683e-02,  5.05128615e-02, -1.42696887e-01, ...,
            4.11595777e-02,  3.64511460e-03,  2.17210472e-01],
          [ 5.33645451e-02,  8.92203972e-02,  3.72816324e-02, ...,
           -3.40342745e-02,  4.44239862e-02, -4.62507317e-03],
          [-3.25056165e-02, -3.90974209e-02, -8.74766055e-03, ...,
            5.45901526e-03, -1.15629798e-02, -7.97724798e-02],
          ...,
          [ 2.78533399e-02,  4.23743911e-02, -6.89867660e-02, ...,
           -1.46777125e-03, -4.94697914e-02,  1.00237310e-01],
          [ 1.84287131e-02,  5.78624429e-04,  6.41532335e-03, ...,
           -1.11485712e-01, -1.44247655e-02, -3.30496021e-02],
          [-4.91377078e-02, -5.26580289e-02,  9.11295414e-03, ...,
           -5.56137133e-03, -2.23566424e-02, -1.17905540e-02]]],


        [[[-3.05062305e-04, -8.34951252e-02, -4.85714152e-02, ...,
            2.64762063e-02, -1.47914484e-01, -8.07159543e-02],
          [-1.08451463e-01, -2.85323057e-03, -5.94426468e-02, ...,
           -5.41105308e-02, -1.64791092e-01, -2.54758522e-02],
          [ 1.53346434e-02, -5.14119677e-02, -8.44290629e-02, ...,
            7.28611648e-02, -1.58833146e-01,  1.54732005e-03],
          ...,
          [ 1.85643196e-01, -3.32637355e-02,  4.33822013e-02, ...,
            1.41427994e-01,  3.40559259e-02, -4.36494499e-02],
          [ 1.03888787e-01, -8.68432149e-02, -4.36970592e-03, ...,
            1.28775015e-01, -2.13370010e-01,  3.61288637e-02],
          [ 6.70932755e-02, -3.43719386e-02, -8.60428065e-02, ...,
            2.17777099e-02, -7.30343070e-03,  1.99916542e-01]],

         [[ 2.68577808e-03, -1.65296569e-01, -9.30643156e-02, ...,
            1.61898863e-02, -1.60984974e-02,  1.51431024e-01],
          [-3.96008082e-02, -8.51563737e-02,  9.59694535e-02, ...,
            4.79491651e-02,  3.03784534e-02,  2.40225047e-02],
          [ 9.27577317e-02, -1.03312410e-01,  2.89471168e-02, ...,
            2.04883926e-02, -9.99920964e-02, -3.45981084e-02],
          ...,
          [ 4.94600274e-02,  1.12857018e-02,  7.65875739e-04, ...,
            7.82735497e-02,  1.41823310e-02,  3.83160189e-02],
          [ 6.18294701e-02,  5.93539886e-02,  4.13541682e-02, ...,
            7.63720796e-02, -1.50781842e-02, -1.46173760e-01],
          [ 2.51172911e-02, -1.23031507e-03, -9.68167558e-03, ...,
           -4.75184955e-02, -3.67983654e-02, -2.34138258e-02]],

         [[ 5.71532287e-02,  2.08238363e-02, -1.22402795e-01, ...,
            6.98505118e-02, -4.27195393e-02, -6.20941892e-02],
          [ 1.94366068e-01,  4.44162041e-02, -8.00787732e-02, ...,
            1.25813941e-02,  3.55778262e-02, -4.10851240e-02],
          [ 1.12232184e-02,  2.80191731e-02,  9.36271797e-04, ...,
           -1.42476559e-02,  1.18864059e-01,  1.26908789e-03],
          ...,
          [ 1.92689613e-01, -1.42471254e-01, -2.74058785e-02, ...,
            1.00844465e-01,  2.39103647e-05,  1.59367975e-02],
          [ 6.80802539e-02,  7.78326914e-02, -3.95599380e-02, ...,
           -4.82397936e-02,  2.47106724e-03, -3.43181426e-04],
          [ 3.84838171e-02, -4.76672687e-02, -2.58279569e-03, ...,
           -2.58013662e-02,  2.60460768e-02, -8.07983428e-03]]],


        [[[-1.61258504e-01, -5.55979982e-02, -4.68061827e-02, ...,
            3.52012552e-03,  4.08665277e-02,  4.84619439e-02],
          [ 4.25779894e-02, -3.85320932e-02,  7.16436803e-02, ...,
            4.98301461e-02, -3.38376649e-02, -1.17614679e-01],
          [ 5.02391271e-02, -6.03627786e-02, -2.78116703e-01, ...,
            9.10190567e-02,  5.21499477e-02,  5.09828515e-02],
          ...,
          [ 6.31624311e-02,  7.11238710e-04, -8.40091035e-02, ...,
           -1.70930643e-02, -1.10101871e-01, -7.66598806e-02],
          [-5.64138368e-02, -1.62609607e-01,  1.05729392e-02, ...,
           -1.15086362e-01,  6.62826821e-02,  1.01914741e-01],
          [ 6.50008172e-02,  3.95820402e-02, -2.99105942e-02, ...,
            8.37561325e-04, -1.26196682e-01,  1.47129055e-02]],

         [[ 1.06566381e-02,  2.50596311e-02,  3.29767801e-02, ...,
           -1.75961964e-02,  1.01140156e-01, -4.69840616e-02],
          [ 6.36988059e-02, -1.17115080e-01, -8.54937639e-03, ...,
           -1.02126345e-01,  7.37119690e-02,  1.54062971e-01],
          [-1.13239698e-01,  4.46358547e-02,  1.69474687e-02, ...,
           -7.08254352e-02,  1.72249317e-01, -1.30635828e-01],
          ...,
          [ 8.33105743e-02, -2.79639848e-02,  1.02350876e-01, ...,
           -4.10773903e-02,  4.38544266e-02,  4.43800427e-02],
          [ 8.83185267e-02,  2.09461991e-02, -8.45748186e-02, ...,
            7.13143572e-02,  1.27832308e-01,  8.16080868e-02],
          [ 1.35318652e-01, -1.30379759e-02, -5.86225130e-02, ...,
            1.37063667e-01, -1.08379558e-01,  3.17332521e-02]],

         [[ 5.86354956e-02, -2.17311867e-02, -6.36831895e-02, ...,
            6.16993345e-02,  6.81424886e-02,  1.07346237e-01],
          [ 1.11819856e-01, -7.19901994e-02,  1.40303463e-01, ...,
           -8.91840458e-02,  1.27518937e-01, -1.28161550e-01],
          [-5.24408072e-02, -1.99108720e-01,  1.07750669e-01, ...,
           -3.35150585e-02, -1.12212591e-01,  7.34768063e-02],
          ...,
          [-4.47433405e-02, -1.58210516e-01,  8.36011693e-02, ...,
            2.15926785e-02,  1.48575321e-01,  9.49218795e-02],
          [ 8.60271901e-02, -7.07514659e-02, -1.00125559e-02, ...,
           -3.96293867e-03,  8.22025239e-02, -5.60843945e-02],
          [-1.22166641e-01, -6.77500591e-02,  6.90614805e-02, ...,
            5.88898994e-02, -6.37848973e-02,  5.41402288e-02]]]],
       dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(16,) dtype=float32, numpy=
 array([-0.00883786, -0.00308503,  0.09293017,  0.15113494,  0.03357508,
         0.02054629, -0.16483617,  0.11283523,  0.03950454,  0.14445296,
         0.13208783, -0.01247809, -0.12735409, -0.02135278, -0.04781489,
         0.08540893], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(16384, 10) dtype=float32, numpy=
 array([[-0.02293865, -0.00456444, -0.00367784, ..., -0.0015067 ,
         -0.00969927,  0.01002026],
        [-0.00504151, -0.00309607, -0.00870905, ...,  0.03004273,
         -0.00429029,  0.01325285],
        [ 0.01299955,  0.01064428,  0.01372442, ..., -0.01928313,
         -0.0263438 , -0.00327954],
        ...,
        [-0.00692748, -0.00961004,  0.00088331, ...,  0.01047849,
          0.01173186,  0.00233957],
        [ 0.00387855,  0.01752277,  0.00528297, ..., -0.01598299,
         -0.00764304,  0.00735474],
        [ 0.00414696, -0.00500712,  0.00966542, ..., -0.00279121,
         -0.01176498, -0.00337222]], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(10,) dtype=float32, numpy=
 array([-0.01379287, -0.05192297,  0.04794208,  0.00754275,  0.06071464,
        -0.01560736,  0.02972453, -0.03510104,  0.01844372, -0.04794323],
       dtype=float32)>]
```

Part V: Train a *GREAT* model on CIFAR-10!

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

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

Describe what you did at the end of the notebook.

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? 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 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?

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

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.

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

Have fun and happy training!

In [39]:

```python
def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1, is_training=False):
    ############################################################################
    # TODO: Train a model on CIFAR-10.                                         #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    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')
```