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

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

%matplotlib inline

def load_cifar10(num_training=49000, num_validation=1000,
num_test=10000):
    """
    Fetch the CIFAR-10 dataset from the web and perform preprocessing
to prepare
    it for the two-layer neural net classifier. These are the same
steps as
```

```
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 dataset and use appropriate data types and
shapes
    cifar10 = tf.keras.datasets.cifar10.load_data()
    (X_train, y_train), (X_test, y_test) = cifar10
    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.int32).flatten()
    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.int32).flatten()

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

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

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

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

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

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-
python.tar.gz
170500096/170498071 [==============================] - 2s 0us/step
170508288/170498071 [==============================] - 2s 0us/step
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

```
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)

# 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.

```
# 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 `placeholders` 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.

```
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))

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.

```
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

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

```
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)*****

    conv_l1 = tf.nn.conv2d(input=x, filters=conv_w1, strides=1,
padding='SAME', data_format='NHWC')
    conv_l1 += conv_b1
    relu_1 = tf.nn.relu(conv_l1)

    conv_l2 = tf.nn.conv2d(input=relu_1, filters=conv_w2, strides=1,
padding='SAME', data_format='NHWC')
    conv_l2 += conv_b2
    relu_2 = tf.nn.relu(conv_l2)

    scores = (flatten(relu_2) @ 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)`.

```python
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

```
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

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


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

```
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.

```
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 = 3.0587
Validation:
     Got 100 / 1000 correct (10.00%)
Epoch 0, iteration 100, loss = 1.7898
Validation:
```

```
     Got 379 / 1000 correct (37.90%)
Epoch 0, iteration 200, loss = 1.5123
Validation:
     Got 399 / 1000 correct (39.90%)
Epoch 0, iteration 300, loss = 1.8223
Validation:
     Got 378 / 1000 correct (37.80%)
Epoch 0, iteration 400, loss = 1.6965
Validation:
     Got 415 / 1000 correct (41.50%)
Epoch 0, iteration 500, loss = 1.7739
Validation:
     Got 438 / 1000 correct (43.80%)
Epoch 0, iteration 600, loss = 1.7530
Validation:
     Got 435 / 1000 correct (43.50%)
Epoch 0, iteration 700, loss = 2.1055
Validation:
     Got 439 / 1000 correct (43.90%)
Epoch 1, iteration 0, loss = 1.4814
Validation:
     Got 433 / 1000 correct (43.30%)
Epoch 1, iteration 100, loss = 1.4485
Validation:
     Got 487 / 1000 correct (48.70%)
Epoch 1, iteration 200, loss = 1.2256
Validation:
     Got 470 / 1000 correct (47.00%)
Epoch 1, iteration 300, loss = 1.5127
Validation:
     Got 441 / 1000 correct (44.10%)
Epoch 1, iteration 400, loss = 1.4369
Validation:
     Got 455 / 1000 correct (45.50%)
Epoch 1, iteration 500, loss = 1.5546
Validation:
     Got 480 / 1000 correct (48.00%)
Epoch 1, iteration 600, loss = 1.5427
Validation:
     Got 459 / 1000 correct (45.90%)
Epoch 1, iteration 700, loss = 1.8043
Validation:
     Got 477 / 1000 correct (47.70%)
Epoch 2, iteration 0, loss = 1.3246
Validation:
     Got 466 / 1000 correct (46.60%)
Epoch 2, iteration 100, loss = 1.3397
Validation:
     Got 497 / 1000 correct (49.70%)
Epoch 2, iteration 200, loss = 1.0883
```

```
Validation:
     Got 487 / 1000 correct (48.70%)
Epoch 2, iteration 300, loss = 1.3649
Validation:
     Got 466 / 1000 correct (46.60%)
Epoch 2, iteration 400, loss = 1.2887
Validation:
     Got 471 / 1000 correct (47.10%)
Epoch 2, iteration 500, loss = 1.4205
Validation:
     Got 496 / 1000 correct (49.60%)
Epoch 2, iteration 600, loss = 1.4306
Validation:
     Got 481 / 1000 correct (48.10%)
Epoch 2, iteration 700, loss = 1.6400
Validation:
     Got 497 / 1000 correct (49.70%)
Epoch 3, iteration 0, loss = 1.2221
Validation:
     Got 488 / 1000 correct (48.80%)
Epoch 3, iteration 100, loss = 1.2546
Validation:
     Got 510 / 1000 correct (51.00%)
Epoch 3, iteration 200, loss = 0.9911
Validation:
     Got 500 / 1000 correct (50.00%)
Epoch 3, iteration 300, loss = 1.2618
Validation:
     Got 470 / 1000 correct (47.00%)
Epoch 3, iteration 400, loss = 1.1719
Validation:
     Got 479 / 1000 correct (47.90%)
Epoch 3, iteration 500, loss = 1.3145
Validation:
     Got 504 / 1000 correct (50.40%)
Epoch 3, iteration 600, loss = 1.3434
Validation:
     Got 489 / 1000 correct (48.90%)
Epoch 3, iteration 700, loss = 1.5200
Validation:
     Got 502 / 1000 correct (50.20%)
Epoch 4, iteration 0, loss = 1.1451
Validation:
     Got 498 / 1000 correct (49.80%)
Epoch 4, iteration 100, loss = 1.1817
Validation:
     Got 516 / 1000 correct (51.60%)
Epoch 4, iteration 200, loss = 0.9072
Validation:
     Got 518 / 1000 correct (51.80%)
```

```
Epoch 4, iteration 300, loss = 1.1774
Validation:
     Got 479 / 1000 correct (47.90%)
Epoch 4, iteration 400, loss = 1.0742
Validation:
     Got 487 / 1000 correct (48.70%)
Epoch 4, iteration 500, loss = 1.2249
Validation:
     Got 506 / 1000 correct (50.60%)
Epoch 4, iteration 600, loss = 1.2674
Validation:
     Got 498 / 1000 correct (49.80%)
Epoch 4, iteration 700, loss = 1.4218
Validation:
     Got 504 / 1000 correct (50.40%)
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.

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

```
Test
     Got 4990 / 10000 correct (49.90%)
```

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.

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

```
    Inputs: None

    Returns a list containing:
    - conv_w1: TensorFlow tf.Variable giving weights for the first
conv layer
    - conv_b1: TensorFlow tf.Variable giving biases for the first conv
layer
    - conv_w2: TensorFlow tf.Variable giving weights for the second
conv layer
    - conv_b2: TensorFlow tf.Variable giving biases for the second
conv layer
    - fc_w: TensorFlow tf.Variable giving weights for the fully-
connected layer
    - fc_b: TensorFlow tf.Variable giving biases for the fully-
connected layer
    """
    params = None

######################################################################
######
    # TODO: Initialize the parameters of the three-layer network.
#

######################################################################
######
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    conv_w1 = tf.Variable(create_matrix_with_kaiming_normal((5, 5, 3,
32)))
    conv_b1 = tf.Variable(tf.zeros((32,)))

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

    fc_w = tf.Variable(create_matrix_with_kaiming_normal((32 * 32 *
16, 10)))
    fc_b = tf.Variable(tf.zeros((10,)))

    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
```

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

```
Epoch 0, iteration 0, loss = 3.4189
Validation:
     Got 119 / 1000 correct (11.90%)
Epoch 0, iteration 100, loss = 1.8535
Validation:
     Got 365 / 1000 correct (36.50%)
Epoch 0, iteration 200, loss = 1.6334
Validation:
     Got 390 / 1000 correct (39.00%)
Epoch 0, iteration 300, loss = 1.7024
Validation:
     Got 393 / 1000 correct (39.30%)
Epoch 0, iteration 400, loss = 1.5952
Validation:
     Got 427 / 1000 correct (42.70%)
Epoch 0, iteration 500, loss = 1.7261
Validation:
     Got 441 / 1000 correct (44.10%)
Epoch 0, iteration 600, loss = 1.6031
Validation:
     Got 465 / 1000 correct (46.50%)
Epoch 0, iteration 700, loss = 1.6178
Validation:
     Got 482 / 1000 correct (48.20%)
Epoch 1, iteration 0, loss = 1.4330
Validation:
     Got 487 / 1000 correct (48.70%)
Epoch 1, iteration 100, loss = 1.4071
Validation:
     Got 494 / 1000 correct (49.40%)
Epoch 1, iteration 200, loss = 1.2830
Validation:
     Got 496 / 1000 correct (49.60%)
Epoch 1, iteration 300, loss = 1.5257
Validation:
     Got 498 / 1000 correct (49.80%)
Epoch 1, iteration 400, loss = 1.3145
Validation:
     Got 515 / 1000 correct (51.50%)
Epoch 1, iteration 500, loss = 1.5413
Validation:
     Got 508 / 1000 correct (50.80%)
Epoch 1, iteration 600, loss = 1.4120
Validation:
     Got 509 / 1000 correct (50.90%)
```

```
Epoch 1, iteration 700, loss = 1.4977
Validation:
     Got 512 / 1000 correct (51.20%)
Epoch 2, iteration 0, loss = 1.2719
Validation:
     Got 531 / 1000 correct (53.10%)
Epoch 2, iteration 100, loss = 1.2766
Validation:
     Got 536 / 1000 correct (53.60%)
Epoch 2, iteration 200, loss = 1.1399
Validation:
     Got 528 / 1000 correct (52.80%)
Epoch 2, iteration 300, loss = 1.4353
Validation:
     Got 541 / 1000 correct (54.10%)
Epoch 2, iteration 400, loss = 1.1686
Validation:
     Got 540 / 1000 correct (54.00%)
Epoch 2, iteration 500, loss = 1.4374
Validation:
     Got 539 / 1000 correct (53.90%)
Epoch 2, iteration 600, loss = 1.3221
Validation:
     Got 545 / 1000 correct (54.50%)
Epoch 2, iteration 700, loss = 1.4188
Validation:
     Got 550 / 1000 correct (55.00%)
Epoch 3, iteration 0, loss = 1.1703
Validation:
     Got 548 / 1000 correct (54.80%)
Epoch 3, iteration 100, loss = 1.1591
Validation:
     Got 555 / 1000 correct (55.50%)
Epoch 3, iteration 200, loss = 1.0333
Validation:
     Got 563 / 1000 correct (56.30%)
Epoch 3, iteration 300, loss = 1.3534
Validation:
     Got 547 / 1000 correct (54.70%)
Epoch 3, iteration 400, loss = 1.0678
Validation:
     Got 571 / 1000 correct (57.10%)
Epoch 3, iteration 500, loss = 1.3569
Validation:
     Got 562 / 1000 correct (56.20%)
Epoch 3, iteration 600, loss = 1.2487
Validation:
     Got 571 / 1000 correct (57.10%)
Epoch 3, iteration 700, loss = 1.3372
Validation:
```

```
     Got 572 / 1000 correct (57.20%)
Epoch 4, iteration 0, loss = 1.0922
Validation:
     Got 575 / 1000 correct (57.50%)
Epoch 4, iteration 100, loss = 1.0550
Validation:
     Got 591 / 1000 correct (59.10%)
Epoch 4, iteration 200, loss = 0.9536
Validation:
     Got 576 / 1000 correct (57.60%)
Epoch 4, iteration 300, loss = 1.2789
Validation:
     Got 556 / 1000 correct (55.60%)
Epoch 4, iteration 400, loss = 0.9963
Validation:
     Got 589 / 1000 correct (58.90%)
Epoch 4, iteration 500, loss = 1.2895
Validation:
     Got 574 / 1000 correct (57.40%)
Epoch 4, iteration 600, loss = 1.1741
Validation:
     Got 577 / 1000 correct (57.70%)
Epoch 4, iteration 700, loss = 1.2544
Validation:
     Got 595 / 1000 correct (59.50%)

[<tf.Variable 'Variable:0' shape=(5, 5, 3, 32) dtype=float32, numpy=
 array([[[[-7.30513420e-04, -8.95354450e-02,  3.45453084e-01, ...,
           -1.60553396e-01, -3.21987778e-01, -2.59407341e-01],
          [ 4.11882192e-01,  8.11211839e-02, -3.84328440e-02, ...,
           -8.51499513e-02, -1.05963789e-01, -6.79349899e-02],
          [-1.00966059e-01,  2.37105880e-02, -1.78850845e-01, ...,
           -1.78173918e-03, -5.41961268e-02, -1.45965377e-02]],

         [[ 4.75981906e-02,  4.68793586e-02,  2.44861066e-01, ...,
            3.75540286e-01,  2.94712204e-02,  1.44733876e-01],
          [-1.72727942e-01, -1.03517100e-01, -1.90017134e-01, ...,
           -4.40343469e-03, -5.14093712e-02, -7.58587271e-02],
          [ 2.61895061e-02, -2.74258375e-01,  2.40599915e-01, ...,
           -1.78961664e-01,  2.22300604e-01,  3.66703480e-01]],

         [[ 1.92213580e-01, -2.20019385e-01,  7.51617029e-02, ...,
           -9.08362046e-02, -6.22506849e-02, -1.28311470e-01],
          [ 1.20961741e-01, -1.80957586e-01, -1.14185281e-01, ...,
           -4.53767449e-01,  7.00392053e-02,  5.55761568e-02],
          [-2.19540313e-01, -3.12285095e-01, -1.90368518e-01, ...,
           -1.58075318e-01, -8.96124840e-02, -1.19667634e-01]],

         [[-2.56883323e-01, -2.70131137e-02,  1.57920673e-01, ...,
            1.77571565e-01,  9.47879851e-02, -7.26439953e-02],
```

```
  [ 1.55969471e-01, -3.36679578e-01, -6.60164282e-02, ...,
    2.43102815e-02, -1.53641522e-01, -1.10994108e-01],
  [-5.77437691e-03, -1.92807958e-01, -2.18520835e-01, ...,
   -3.50708254e-02, -1.78002268e-01,  1.75687432e-01]],

 [[-3.09088230e-01, -5.29281469e-03,  7.97499269e-02, ...,
    4.39973399e-02,  1.15351386e-01,  2.31265038e-01],
  [-4.36196662e-02,  9.00692120e-02,  1.09408498e-01, ...,
   -2.78051347e-01, -1.59723982e-01,  1.34843960e-01],
  [ 5.38315699e-02, -1.38887092e-01, -1.72834516e-01, ...,
   -1.24663949e-01, -4.01846580e-02,  7.67067447e-02]]],


[[[ 9.60446969e-02,  1.75726324e-01, -2.36976117e-01, ...,
    1.30022973e-01, -1.36510044e-01, -1.52397677e-01],
  [-8.21835082e-03,  4.28797379e-02, -9.00332034e-02, ...,
    1.05719775e-01, -1.04075879e-01,  7.36046433e-02],
  [ 8.57616514e-02,  4.00555618e-02, -1.80147946e-01, ...,
   -5.30982949e-02, -2.59078573e-02,  2.49687612e-01]],

 [[ 2.25261480e-01, -5.63800223e-02,  3.27074140e-01, ...,
   -1.37878299e-01, -1.26921549e-01,  3.30219805e-01],
  [ 2.06702054e-01, -1.66422293e-01,  1.54591858e-01, ...,
   -2.20067993e-01, -4.40400727e-02, -1.00898862e-01],
  [ 2.38133088e-01, -1.13479123e-01,  7.11209625e-02, ...,
   -2.51086235e-01, -9.99978259e-02,  5.94347075e-04]],

 [[-2.35511243e-01, -1.81210905e-01,  3.52863371e-01, ...,
    9.49818119e-02, -9.88538861e-02, -1.04743347e-01],
  [ 2.47491047e-01, -2.00806782e-01,  1.27114952e-01, ...,
   -5.05219549e-02, -1.10118464e-01, -3.32692415e-02],
  [-1.90347642e-01, -1.63624451e-01,  1.85606271e-01, ...,
   -2.55194813e-01,  2.14366212e-01,  5.08317314e-02]],

 [[ 1.50774149e-02, -1.66397125e-01,  1.23035505e-01, ...,
   -1.98445871e-01,  1.46452099e-01, -9.87487584e-02],
  [ 1.42581701e-01,  2.27972418e-01, -1.02373809e-01, ...,
   -2.38618508e-01, -3.04274727e-02,  1.43364087e-01],
  [-2.71013193e-02, -3.52833383e-02,  7.23565146e-02, ...,
    2.29501188e-01,  3.11383933e-01,  1.10835567e-01]],

 [[-6.27629384e-02,  2.39807487e-01, -1.92859575e-01, ...,
   -1.08327784e-01,  1.52276427e-01,  7.58924931e-02],
  [ 1.99909702e-01, -5.43549582e-02,  3.16712796e-03, ...,
   -3.05295736e-02,  7.14420304e-02, -3.08946632e-02],
  [-1.45296067e-01,  3.66693437e-02,  9.19080675e-02, ...,
   -3.66911232e-01,  1.99497759e-01, -1.94348797e-01]]],


[[[-8.47436190e-02, -2.12281898e-01, -2.01900274e-01, ...,
```

```
   2.92543638e-02,  1.64096355e-01, -4.00138088e-02],
 [-2.71748781e-01,  2.13719264e-01,  4.06061709e-02, ...,
  -8.94356743e-02, -4.06868756e-01, -9.59772989e-02],
 [ 7.25255832e-02, -9.32211876e-02, -2.28253648e-01, ...,
   2.56212037e-02, -1.42037362e-01,  1.90084100e-01]],

[[-2.98245866e-02,  1.98042095e-01,  3.13281231e-02, ...,
   4.77557443e-02,  5.50047047e-02, -1.75231174e-02],
 [-1.38513461e-01,  2.51012057e-01, -1.36301070e-02, ...,
   2.80899890e-02, -3.87172341e-01, -7.15483055e-02],
 [-2.37749621e-01, -8.51549283e-02,  4.49798964e-02, ...,
  -1.56762585e-01, -7.42390156e-02,  8.46393332e-02]],

[[-2.58509278e-01, -9.01283473e-02,  1.83262214e-01, ...,
  -2.40244001e-01,  3.03251952e-01, -1.33128926e-01],
 [-3.69378887e-02, -2.04483539e-01, -2.18099624e-01, ...,
  -1.53684439e-02, -1.17457710e-01, -3.43149930e-01],
 [-6.02870919e-02, -2.11062998e-01, -8.17419142e-02, ...,
   2.95075715e-01, -2.78160870e-01, -1.43335298e-01]],

[[-1.40570700e-01,  1.35524720e-01,  1.37879044e-01, ...,
   4.12743062e-01,  1.81679547e-01, -4.99341078e-02],
 [-4.16073054e-01,  2.21707746e-01, -8.72646347e-02, ...,
   7.22639412e-02,  1.02528920e-02, -8.71508494e-02],
 [ 1.35479514e-02,  3.43043171e-02,  2.17093732e-02, ...,
   2.33303308e-01,  8.81786421e-02, -3.38994950e-01]],

[[ 4.14320678e-01,  4.70753014e-02, -8.70797634e-02, ...,
   1.10166118e-01, -7.03277141e-02,  1.93497948e-02],
 [-3.62597988e-03, -1.35737183e-02,  4.40387242e-02, ...,
  -2.27616988e-02, -1.66044980e-01,  1.38709202e-01],
 [-1.79226071e-01, -3.10987055e-01, -2.30489731e-01, ...,
  -2.74551716e-02,  3.55695516e-01, -8.84238780e-02]]],


[[[ 2.92637557e-01,  1.11136720e-01, -1.73627123e-01, ...,
  -7.59075284e-02, -2.13911116e-01, -9.06124935e-02],
 [ 2.53636837e-01,  3.08935083e-02,  5.74205406e-02, ...,
   7.90975168e-02, -5.68337366e-02,  2.38036007e-01],
 [ 1.25837082e-03,  1.47690013e-01, -9.72679630e-02, ...,
  -1.04361484e-02, -2.45479345e-02, -1.47123113e-01]],

[[-2.58654565e-01,  3.18611860e-02,  2.76930660e-01, ...,
  -1.80394948e-02,  1.71229124e-01, -2.65229613e-01],
 [-1.75435066e-01, -2.87006527e-01, -1.44984282e-04, ...,
   2.43524492e-01,  1.65802732e-01,  2.22214200e-02],
 [ 1.25075832e-01,  1.31595090e-01,  2.64160931e-01, ...,
   1.90979883e-01, -7.93517381e-03,  8.32797587e-02]],

[[ 7.26914853e-02,  1.96674034e-01, -1.56220138e-01, ...,
```

```
  -3.36369783e-01, -2.61677504e-02, -4.49772365e-03],
 [ 8.11844692e-02, -1.72738969e-01,  2.64287256e-02, ...,
  -3.58128995e-01, -1.47813037e-02, -3.13234568e-01],
 [ 1.33717492e-01,  5.39439246e-02,  6.59105256e-02, ...,
   1.64549321e-01, -5.80490269e-02,  1.13754161e-01]],

[[-2.05189720e-01,  8.64141136e-02,  9.74737406e-02, ...,
   3.25543433e-02,  2.54794687e-01, -2.27786228e-01],
 [-6.91001415e-02,  1.73223078e-01, -1.95502505e-01, ...,
   1.06233716e-01,  1.08838044e-01,  7.96907917e-02],
 [ 1.60418242e-01, -7.48278107e-03, -1.35423884e-01, ...,
   3.55620421e-02, -5.85079677e-02, -2.47734133e-02]],

[[ 5.41471317e-02,  2.29570910e-01, -1.24866348e-02, ...,
   6.44945428e-02,  1.72648802e-01, -8.58603418e-02],
 [ 8.79785046e-02,  1.04059853e-01,  1.88528329e-01, ...,
   1.81184430e-02,  8.44091773e-02, -1.58846691e-01],
 [ 1.64371386e-01, -4.89250794e-02,  1.42225400e-01, ...,
   1.02799952e-01,  1.19101286e-01, -1.93702474e-01]]],


[[[ 1.21607982e-01,  1.44058704e-01,  2.28517443e-01, ...,
  -7.37447217e-02, -1.08264521e-01, -2.08040938e-01],
 [-1.37451515e-01, -2.13544235e-01, -1.62027985e-01, ...,
   2.63097048e-01, -3.02944392e-01,  2.65635282e-01],
 [ 6.56668842e-02,  1.84970982e-02,  1.64642617e-01, ...,
   5.68933897e-02, -8.63237977e-02, -4.89637628e-02]],

[[-2.88903769e-02, -4.96029966e-02,  4.03755046e-02, ...,
  -3.13823968e-02,  1.30422160e-01, -9.75670740e-02],
 [-4.95196208e-02, -4.23092321e-02,  2.05306262e-01, ...,
   5.96750565e-02,  1.44636452e-01,  1.48844525e-01],
 [-3.74313146e-01,  2.24484071e-01, -1.74161032e-01, ...,
   1.14845827e-01,  1.18584670e-01,  3.10371667e-02]],

[[ 2.08580151e-01,  6.19445667e-02, -2.27674797e-01, ...,
  -8.62284452e-02,  7.28449598e-02,  1.62741646e-01],
 [-1.03527606e-01, -7.33649060e-02, -1.16528094e-01, ...,
   4.85028811e-02,  2.71041766e-02,  1.23176575e-01],
 [-1.71095282e-01, -2.00082913e-01, -2.17602365e-02, ...,
   1.63100958e-01, -2.07137600e-01, -4.57553118e-02]],

[[ 1.10032380e-01,  1.08166680e-01,  4.71700355e-02, ...,
  -5.67834899e-02, -3.89138237e-02,  1.67086318e-01],
 [-1.45644873e-01,  1.81135088e-01, -7.00335726e-02, ...,
  -1.73095446e-02,  2.31738925e-01, -1.42796874e-01],
 [ 1.78415384e-02, -1.04858555e-01, -1.43666998e-01, ...,
   2.18027651e-01,  4.59059216e-02, -2.58112196e-02]],

[[ 1.99003801e-01, -1.57329738e-01, -1.17594413e-01, ...,
```

```
          -7.89504033e-03, -2.72670835e-01, -2.12593913e-01],
         [-4.25807014e-02,  1.55736819e-01, -1.07949063e-01, ...,
           2.77779877e-01, -1.05633646e-01,  1.65189598e-02],
         [ 1.38120204e-01,  3.78111899e-02, -5.36197610e-02, ...,
           1.27752915e-01, -3.87001596e-02, -2.95460150e-02]]]],
      dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(32,) dtype=float32, numpy=
 array([-0.02768716,  0.00223203,  0.01527918, -0.06612939,
0.10072886,
         0.13191624,  0.02359801, -0.01461088,  0.01141547,
0.10225188,
        -0.09295668,  0.01785557, -0.00898188, -0.06068878,
0.06157703,
        -0.01063985,  0.03661035,  0.05755736, -0.00181908,
0.01993922,
        -0.01384589, -0.03255774,  0.02717302,  0.03633253,
0.00224985,
         0.0004818 , -0.01037123, -0.05592122, -0.01850329, -
0.07245331,
        -0.0272992 ,  0.00288302], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(3, 3, 32, 16) dtype=float32, numpy=
 array([[[[ 1.20059803e-01,  3.54150496e-02, -1.16539448e-01, ...,
           7.32756630e-02,  1.49674535e-01,  7.59213194e-02],
         [-1.54817319e-02, -1.10467663e-02,  4.77048978e-02, ...,
           8.84246230e-02, -2.91190922e-01, -6.03775457e-02],
         [-4.22001556e-02,  1.04407974e-01,  8.30594972e-02, ...,
           6.23203488e-03,  7.37971021e-03, -5.81124332e-04],
         ...,
         [-2.98400186e-02,  3.29618454e-02, -4.68634628e-02, ...,
           4.41521779e-02, -8.50757509e-02,  5.10611497e-02],
         [ 1.74496010e-01, -2.00748593e-01, -8.62521827e-02, ...,
          -1.17483819e-02, -1.35560244e-01, -1.31330401e-01],
         [-1.42517716e-01,  1.21640876e-01,  2.33854223e-02, ...,
          -4.55923267e-02, -4.54068147e-02, -1.06986091e-01]],

        [[-5.09446822e-02, -4.01116982e-02, -7.16414899e-02, ...,
          -7.77624473e-02,  6.51369244e-02, -1.23899810e-01],
         [-1.67219281e-01, -4.92069200e-02, -4.40299213e-02, ...,
          -3.95301618e-02,  9.22825709e-02,  5.81754595e-02],
         [ 7.87347406e-02,  2.68691927e-02,  2.18210015e-02, ...,
          -2.42666639e-02, -3.01675219e-02, -1.07080467e-01],
         ...,
         [ 1.67581178e-02,  4.00529876e-02, -3.35627571e-02, ...,
          -1.60332367e-01, -1.60853133e-01, -1.32263616e-01],
         [-4.92318720e-02,  5.22663333e-02, -1.88266471e-01, ...,
           1.14093171e-02,  1.18283965e-02, -7.53860921e-02],
         [ 5.98984249e-02, -2.49389783e-02, -8.74560848e-02, ...,
           1.35376565e-02,  1.34119019e-02,  5.45621244e-03]],

        [[ 2.18049549e-02, -6.85680881e-02, -8.93224999e-02, ...,
```

```
   -1.30685583e-01,  1.49014175e-01, -1.67732030e-01],
  [-1.68711524e-02, -3.67785357e-02,  1.30541742e-01, ...,
   -4.12533656e-02,  1.00530528e-01, -6.40092492e-02],
  [-4.23601046e-02,  8.68271366e-02,  5.49914613e-02, ...,
    1.26764253e-01,  9.24714953e-02, -6.35858476e-02],
  ...,
  [ 2.82024592e-02,  1.13337770e-01, -8.22410807e-02, ...,
   -1.78071801e-02,  5.54744788e-02,  6.86601270e-03],
  [ 1.02904774e-01, -3.28449123e-02, -6.95606768e-02, ...,
    6.70144632e-02,  2.14794263e-01,  1.02457702e-01],
  [-7.48260319e-02,  1.66375786e-01, -4.80165184e-02, ...,
   -7.36360550e-02,  6.50548860e-02, -2.25106344e-01]]],


[[[ 4.62797396e-02,  1.31813325e-02, -1.07579440e-01, ...,
   -1.30745590e-01,  7.26507455e-02,  3.84357311e-02],
  [-1.32609168e-02,  6.27964512e-02, -1.35661900e-01, ...,
    6.69100508e-03, -4.35005277e-02,  7.84188043e-03],
  [-3.73819955e-02,  5.30885942e-02,  3.11627761e-02, ...,
   -3.97149064e-02,  1.04513634e-02,  3.84740718e-02],
  ...,
  [-1.17947005e-01, -2.53796913e-02, -1.50833875e-01, ...,
    3.40880528e-02,  8.57422948e-02,  9.22607929e-02],
  [-2.27083907e-01, -6.23057038e-03,  4.71546547e-03, ...,
    9.67583135e-02, -5.04814051e-02, -5.21458238e-02],
  [ 5.73384687e-02,  4.25049327e-02, -5.22212982e-02, ...,
   -8.19359496e-02, -1.99814513e-02, -1.96005851e-02]],

 [[ 7.02460334e-02, -3.30689773e-02,  1.72584832e-01, ...,
    6.14651255e-02,  2.12885857e-01,  1.25933230e-01],
  [-1.10953785e-01, -7.25947767e-02, -5.86223044e-02, ...,
   -6.27389252e-02, -1.06549216e-02,  1.91470869e-02],
  [-9.24827829e-02, -1.68501064e-02,  4.79414448e-04, ...,
    1.94409862e-02, -9.31788236e-02,  9.38434270e-04],
  ...,
  [ 7.87438676e-02,  1.19018294e-01, -1.27002969e-01, ...,
   -5.77672049e-02, -1.73113927e-01, -8.73927400e-02],
  [ 7.88122118e-02, -1.29690796e-01, -8.38287324e-02, ...,
   -3.52056287e-02,  2.21782122e-02,  9.71581116e-02],
  [ 1.10432670e-01,  4.55881245e-02,  3.43543328e-02, ...,
   -9.20929462e-02, -5.00425100e-02,  6.18262403e-02]],

 [[-6.58216029e-02,  1.12133577e-01, -6.28019273e-02, ...,
    3.11390050e-02, -6.38870895e-02,  3.32061574e-02],
  [-4.39621620e-02, -2.48275287e-02, -1.99584216e-02, ...,
    1.48920044e-01,  8.26861151e-03,  4.64214245e-03],
  [ 1.04204059e-01,  7.16759712e-02, -4.18848917e-02, ...,
    8.35831985e-02, -6.31407648e-03,  7.46269226e-02],
  ...,
  [-1.12159662e-01,  1.67022515e-02,  4.01990265e-02, ...,
```

```
         -8.25469047e-02,  2.44262461e-02,  1.23848274e-01],
        [-7.31497929e-02,  9.39674489e-03,  2.63151407e-01, ...,
         -9.86544564e-02, -2.74871811e-02, -1.87037513e-02],
        [ 8.34036544e-02,  1.70183167e-01, -8.46275315e-02, ...,
         -7.94818550e-02, -7.09031671e-02,  1.15632480e-02]]],


      [[[ 1.93232313e-01, -8.66758898e-02, -1.00858770e-01, ...,
         -6.79915622e-02,  3.22237611e-01, -5.43866074e-03],
        [-6.54150173e-02,  3.53442854e-03,  4.96104993e-02, ...,
         -2.86888480e-02, -7.78877065e-02,  4.14807461e-02],
        [ 8.63576829e-02,  2.90804505e-02,  5.94693869e-02, ...,
          7.44697228e-02, -1.71435550e-01, -4.86574247e-02],
        ...,
        [-1.57211944e-02,  4.25882600e-02, -1.30633757e-01, ...,
         -2.69780345e-02, -1.30904183e-01,  1.07615486e-01],
        [-3.58388871e-02, -1.28015190e-01, -7.51438886e-02, ...,
          9.85277351e-03,  9.38114226e-02,  6.53349310e-02],
        [-2.81950459e-02,  9.40734241e-03, -1.27954157e-02, ...,
          7.89094642e-02,  6.36389777e-02,  2.26675496e-01]],

       [[-2.86152940e-02, -2.61857081e-02,  1.57317426e-02, ...,
          2.92053837e-02,  2.22676888e-01, -4.61234674e-02],
        [-8.39840472e-02,  5.74816437e-03,  3.60342041e-02, ...,
         -1.51291862e-01, -1.16554849e-01,  4.18216437e-02],
        [-5.57556972e-02, -9.60214238e-05, -1.30307795e-02, ...,
          6.16984963e-02, -1.74279884e-01, -5.98868355e-02],
        ...,
        [-3.85917127e-02,  5.69658652e-02,  2.72334479e-02, ...,
         -8.21942464e-03, -2.37414744e-02, -2.97677666e-02],
        [-7.61016309e-02, -1.09260187e-01, -6.87472289e-03, ...,
          1.79968309e-02, -7.68606514e-02,  8.63021016e-02],
        [ 1.28589824e-01, -4.60620113e-02, -6.39083534e-02, ...,
         -1.34251565e-01,  3.35333534e-02,  5.22671379e-02]],

       [[ 1.61902770e-01, -9.99846756e-02,  1.36351751e-04, ...,
         -6.12963624e-02, -7.12286904e-02, -3.00347013e-03],
        [-2.65037809e-02,  1.47279706e-02,  1.16906926e-01, ...,
         -5.04875518e-02,  3.13043967e-02, -1.33030400e-01],
        [ 8.49244595e-02, -1.42229721e-01, -4.65722792e-02, ...,
          5.29998019e-02,  4.17467058e-02, -2.44385451e-02],
        ...,
        [ 1.16433710e-01,  6.91298246e-02,  8.97793993e-02, ...,
         -5.42396232e-02,  8.01123157e-02,  1.70534197e-02],
        [ 1.76667109e-01, -1.53340712e-01,  3.04887928e-02, ...,
         -3.34824435e-02, -1.02740310e-01, -5.86066768e-02],
        [ 4.90034148e-02, -6.08791523e-02,  8.78448486e-02, ...,
          2.52332147e-02, -7.67879933e-02,  5.93101345e-02]]]],
      dtype=float32)>,
<tf.Variable 'Variable:0' shape=(16,) dtype=float32, numpy=
```

```
 array([-0.08106682,  0.04419798,  0.15957175,  0.01627744,
0.00672597,
         0.02422851,  0.0102271 , -0.01056297, -0.00089165, -
0.19989067,
        -0.00421291,  0.01291813,  0.0051431 ,  0.26594704,
0.12179016,
        -0.02360822], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(16384, 10) dtype=float32, numpy=
 array([[-0.0039793 ,  0.0149017 ,  0.00580458, ...,  0.00256496,
          0.01397877, -0.0067488 ],
        [-0.01815603,  0.00296218,  0.00309442, ..., -0.00925748,
          0.0165246 ,  0.00727398],
        [ 0.00275394, -0.00412423, -0.0014877 , ...,  0.01535041,
          0.01270547, -0.01130752],
        ...,
        [ 0.00213022, -0.00551291,  0.00072743, ...,  0.0020359 ,
          0.00602596,  0.00078809],
        [ 0.0008564 ,  0.01645213, -0.00925645, ..., -0.00134157,
          0.00491678, -0.01932548],
        [-0.01443114, -0.01230106, -0.00764458, ...,  0.01654562,
         -0.00104443, -0.00508382]], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(10,) dtype=float32, numpy=
 array([-0.01043312, -0.04917646,  0.04214871,  0.00635859,
0.05130031,
        -0.01271202,  0.01986577, -0.02561304,  0.02710512, -
0.04884399],
       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!

```
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)*****
  if is_training:
    model = model_init_fn()
    optimizer = optimizer_init_fn()
    model.compile(optimizer, loss="sparse_categorical_crossentropy",
metrics=['accuracy'])
    model.fit(X_train, y_train, batch_size=64, epochs=num_epochs,
validation_data=(X_val, y_val), shuffle=True)
  return model

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

# With reference to kaggle: https://www.kaggle.com/vakninmaor/cifar-
10-for-beginners-score-90

from tensorflow import keras
from keras import layers

class CustomConvNet(tf.keras.Model):
    def __init__(self):
        super(CustomConvNet, self).__init__()

#################################################################
######
        # TODO: Construct a model that performs well on CIFAR-10
#
```

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

        self.conv1 = layers.Conv2D(32, (3, 3), activation='relu',
strides=(1,1), padding='same', input_shape=(32, 32, 3))
        self.bn1 = layers.BatchNormalization()
        self.dc1 = layers.DepthwiseConv2D(kernel_size=(3,3),
strides=(1, 1), padding='same', activation=keras.activations.relu,
depth_multiplier=3)
        self.dp1 = layers.Dropout(rate=0.1)

        self.conv2 = layers.Conv2D(64, (3, 3), activation='relu',
strides=(2, 2), padding='same')
        self.bn2 = layers.BatchNormalization()
        self.dc2 = layers.DepthwiseConv2D(kernel_size=(3,3),
strides=(1, 1), padding='same', activation=keras.activations.relu)
        self.dp2 = layers.Dropout(rate=0.1)

        self.conv3 = layers.Conv2D(128, (3, 3), activation='relu',
strides=(1, 1), padding='same')
        self.bn3 = layers.BatchNormalization()
        self.dc3 = layers.DepthwiseConv2D(kernel_size=(3,3),
strides=(1, 1), padding='same', activation=keras.activations.relu)
        self.dp3 = layers.Dropout(rate = 0.4)

        self.conv4 = layers.Conv2D(128, (3, 3), activation='relu',
strides=(1, 1), padding='same')
        self.bn4 = layers.BatchNormalization()
        self.dc4 = layers.DepthwiseConv2D(kernel_size=(1,1),
strides=(1, 1), padding='same', activation=keras.activations.relu)


        self.conv5 = layers.Conv2D(256, (3, 3), activation='relu',
strides=(2, 2), padding='same')
        self.bn5 = layers.BatchNormalization()
        self.dc5 = layers.DepthwiseConv2D(kernel_size=(3,3),
strides=(1, 1), padding='same', activation=keras.activations.relu)


        self.conv6 = layers.Conv2D(512, (1, 1), activation='relu',
strides=(2, 2), padding='same')
        self.bn6 = layers.BatchNormalization()
        self.dc6 = layers.DepthwiseConv2D(kernel_size=(1,1),
strides=(1, 1), padding='same', activation=keras.activations.relu)
        self.dp4 = layers.Dropout(rate = 0.4)

        self.flat = layers.Flatten()
        self.den1 = layers.Dense(2048, activation='relu')
```

```
        self.den2 = layers.Dense(512, activation='relu')
        self.den3 = layers.Dense(10, activation='softmax')

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

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

######################################################################
######

    def call(self, input_tensor, training=False):

######################################################################
######
        # TODO: Construct a model that performs well on CIFAR-10
#

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

        output = self.conv1(input_tensor)
        output = self.bn1(output)
        output = self.dc1(output)
        output = self.dp1(output)

        output = self.conv2(output)
        output = self.bn2(output)
        output = self.dc2(output)
        output = self.dp2(output)

        output = self.conv3(output)
        output = self.bn3(output)
        output = self.dc3(output)
        output = self.dp3(output)

        output = self.conv4(output)
        output = self.bn4(output)
        output = self.dc4(output)


        output = self.conv5(output)
        output = self.bn5(output)
        output = self.dc5(output)
```

```
        output = self.conv6(output)
        output = self.bn6(output)
        output = self.dc6(output)
        output = self.dp4(output)

        output = self.flat(output)
        output = self.den1(output)
        output = self.den2(output)
        output = self.den3(output)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

######################################################################
######

        return output

device = '/device:GPU:0'   # Change this to a CPU/GPU as you wish!
# device = '/cpu:0'        # Change this to a CPU/GPU as you wish!
print_every = 700
num_epochs = 10

model = CustomConvNet()

def model_init_fn():
    return CustomConvNet()

def optimizer_init_fn():
    learning_rate = 1e-3
    return tf.keras.optimizers.Adam(learning_rate)
print('Train')
trained_params =train_part34(model_init_fn, optimizer_init_fn,
num_epochs=num_epochs, is_training=True)

model = model_init_fn()
print('Test')
score = trained_params.evaluate(X_test, y_test)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train
Epoch 1/10
766/766 [==============================] - 1219s 1s/step - loss:
1.2864 - accuracy: 0.5278 - val_loss: 0.9793 - val_accuracy: 0.6560
Epoch 2/10
```

```
766/766 [==============================] - 1128s 1s/step - loss:
0.8414 - accuracy: 0.7031 - val_loss: 0.8379 - val_accuracy: 0.7140
Epoch 3/10
766/766 [==============================] - 1147s 1s/step - loss:
0.6818 - accuracy: 0.7613 - val_loss: 0.5882 - val_accuracy: 0.8020
Epoch 4/10
766/766 [==============================] - 1135s 1s/step - loss:
0.5713 - accuracy: 0.7993 - val_loss: 0.7006 - val_accuracy: 0.7600
Epoch 5/10
766/766 [==============================] - 1125s 1s/step - loss:
0.4949 - accuracy: 0.8275 - val_loss: 0.5874 - val_accuracy: 0.8090
Epoch 6/10
766/766 [==============================] - 1130s 1s/step - loss:
0.4263 - accuracy: 0.8488 - val_loss: 0.5389 - val_accuracy: 0.8240
Epoch 7/10
766/766 [==============================] - 1131s 1s/step - loss:
0.3742 - accuracy: 0.8676 - val_loss: 0.5823 - val_accuracy: 0.8340
Epoch 8/10
766/766 [==============================] - 1128s 1s/step - loss:
0.3168 - accuracy: 0.8872 - val_loss: 0.5090 - val_accuracy: 0.8350
Epoch 9/10
766/766 [==============================] - 1129s 1s/step - loss:
0.2725 - accuracy: 0.9031 - val_loss: 0.5966 - val_accuracy: 0.8330
Epoch 10/10
766/766 [==============================] - 1144s 1s/step - loss:
0.2375 - accuracy: 0.9166 - val_loss: 0.7666 - val_accuracy: 0.7970
Test
313/313 [==============================] - 55s 177ms/step - loss:
0.7076 - accuracy: 0.7982
Test loss: 0.7076130509376526
Test accuracy: 0.7982000112533569
```

Describe what you did

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

This neural network has 12 convolution blocks. six convolution blocks are normal convolution and the other six are depthwise convolution. Between each convolution block, batch normalization is used to 'reset' the distribution of the output of the previous layer to be more efficiently processed by the subsequent layer. To avoid overfitting, a dropout is used at the end of 2 convolution blocks.

After the 12 convolution blocks, the flatten is used to convert the 4D inputs into 2D which is required for the next fully-connected dense layer. Two dense ReLU layers are used to combine the layers and followed by a final softmax layer for classification. This final layer contains probabilities of the 10 classes in which the sum is 1 and the argmax would be the predicted label.

Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from libs.classifiers.cnn import *
from libs.data_utils import get_CIFAR10_data
from libs.gradient_check import eval_numerical_gradient_array,
eval_numerical_gradient
from libs.layers import *
from libs.fast_layers import *
from libs.solver import Solver

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

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

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

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

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `libs/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

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

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

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

```
Testing conv_forward_naive

---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetwork
```

```
s.ipynb Cell 5' in <cell line: 24>()
     <a href='vscode-notebook-cell:/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb#ch0000004?line=21'>22</a> # Compare your output to ours; difference should be around e-8
     <a href='vscode-notebook-cell:/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb#ch0000004?line=22'>23</a> print('Testing conv_forward_naive')
---> <a href='vscode-notebook-cell:/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb#ch0000004?line=23'>24</a> print('difference: ', rel_error(out, correct_out))

/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb Cell 2' in rel_error(x, y)
     <a href='vscode-notebook-cell:/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb#ch0000001?line=20'>21</a> def rel_error(x, y):
     <a href='vscode-notebook-cell:/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb#ch0000001?line=21'>22</a>   """ returns relative error """
---> <a href='vscode-notebook-cell:/Users/yuanhawk/50.035-Computer-Vision/Lab2/week6/ConvolutionalNetworks.ipynb#ch0000001?line=22'>23</a>   return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

TypeError: unsupported operand type(s) for -: 'NoneType' and 'float'
```

Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

```
from imageio import imread
from PIL import Image

kitten = imread('notebook_images/kitten.jpg')
puppy = imread('notebook_images/puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_puppy = np.array(Image.fromarray(puppy).resize((img_size, img_size)))
resized_kitten = np.array(Image.fromarray(kitten_cropped).resize((img_size, img_size)))
```

```
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = resized_puppy.transpose((2, 0, 1))
x[1, :, :, :] = resized_kitten.transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in
w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_no_ax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_no_ax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_no_ax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_no_ax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_no_ax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
```

```
imshow_no_ax(out[1, 1])
plt.show()
```

Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `libs/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x,
w, b, conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x,
w, b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x,
w, b, conv_param)[0], b, dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

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

Max-Pooling: Naive forward

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

Check your implementation by running the following:

```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)
```

```
correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

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

Max-Pooling: Naive backward

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

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

```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

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

Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `libs/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `libs` directory:

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

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

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

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

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

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

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
```

```
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))

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

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

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

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

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

Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `libs/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check they're working.

```
from libs.layer_utils import conv_relu_pool_forward,
conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
```

```
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

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

```
from libs.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x,
w, b, conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x,
w, b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x,
w, b, conv_param)[0], b, dout)

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

Three-layer ConvNet

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

Open the file `libs/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

Sanity check loss

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

```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)
```

Gradient check

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

```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, 
model.params[param_name], verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
```

```
    print('%s max relative error: %e' % (param_name,
rel_error(param_grad_num, grads[param_name])))
```

Overfit small data

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

```
np.random.seed(231)

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

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=15, batch_size=50,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

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

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

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

Train the net

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

```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500,
reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

Visualize Filters

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

```
from libs.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```

Dropout

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

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

Acknowledgement: This exercise is adapted from Stanford CS231n.

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from libs.classifiers.fc_net import *
from libs.data_utils import get_CIFAR10_data
from libs.gradient_check import eval_numerical_gradient,
eval_numerical_gradient_array
from libs.solver import Solver

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

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

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

# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

Dropout forward pass

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

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

```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

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

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

Dropout backward pass

In the file `libs/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

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

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,
dropout_param)[0], x, dout)

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

Fully-connected nets with Dropout

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

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

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

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

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

Regularization experiment

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

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

solvers = {}
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)
```

```
  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='sgd',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
  print()

# Plot train and validation accuracies of the two models

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

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

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

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

Question

Explain what you see in this experiment. What does it suggest about **dropout**?

Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

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

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

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

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

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

  return dx, dw
```

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

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

Acknowledgement: This exercise is adapted from Stanford CS231n.

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from libs.classifiers.fc_net import *
from libs.data_utils import get_CIFAR10_data
from libs.gradient_check import eval_numerical_gradient,
eval_numerical_gradient_array
from libs.solver import Solver

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

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

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

# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

Affine layer: foward

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

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

```
# Test the affine_forward function

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

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

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

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

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

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

Affine layer: backward

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

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

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

```
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout)

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

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  1.0908199508708189e-10
dw error:  2.1752492052093605e-10
db error:  1.8810031119556898e-11
```

ReLU activation: forward

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

```
# Test the relu_forward function

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

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

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

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

ReLU activation: backward

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

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

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

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

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

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

"Sandwich" layers

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

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

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

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

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

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax

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

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

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

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

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

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

Two-layer network

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

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

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

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

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,
14.57198434, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,
14.81149128, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,
15.05099822, 15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

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

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

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-08
W2 relative error: 3.45e-10
b1 relative error: 8.01e-09
b2 relative error: 2.53e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 1.37e-07
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

Solver

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

Open the file `libs/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

```
# X_val:  (1000, 3, 32, 32)
# X_train:  (49000, 3, 32, 32)
# X_test:  (1000, 3, 32, 32)
# y_val:  (1000,)
# y_train:  (49000,)
# y_test:  (1000,)

# model = TwoLayerNet()
# solver = None

######################################################################
########
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at
least  #
# 50% accuracy on the validation set.
#
######################################################################
```

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

solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                    'learning_rate': 1e-3,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
solver.train()

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

# Run this cell to visualize training loss and train / val accuracy

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

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

Multilayer network

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

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

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

Initial loss and gradient check

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

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

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

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2,
dtype=np.float64)

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

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-
5
  # for the check when reg = 0.0
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name],
verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num,
grads[name])))
```

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

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

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

weight_scale = 1e-2   # Experiment with this!
learning_rate = 1e-4  # Experiment with this!
```

```
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

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