CIFAR-10 Classification with Convolutional Neural Networks

Daniel Petti

March 2017

1 Introduction

The goal of this document is to provide instructions for constructing a simple CNN-based CIFAR-10 classifier. This includes instructions for acquiring and manipulating the dataset, as well as some helpful tips for model construction, training, and evaluation.

1.1 What is CIFAR-10?

CIFAR-10 is a dataset consisting of 32x32 color images which fall into one of ten categories: "airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck". There is no overlap between any of the categories. CIFAR-10 provides a set of 50,000 training images, and 10,000 testing images. For the purposes of this exercise, 5000 of the testing images will be withheld for final evaluation purposes, and therefore, will not be made available to the students.

1.2 What can I do with it?

Your task for this assignment is to implement and train a CIFAR-10 classifier using TensorFlow. The TensorFlow documentation already includes a example CIFAR 10 classifier, however, the model we will be making will use a different structure. (This model sacrifices some performance relative to the example one in order to reduce the training time on student's computers.)

2 Using the Dataset

2.1 Downloading

For your convenience, we have already provided the CIFAR-10 data that you need, pre-packaged into a single file. It is available for download here. It is stored already in a Pickle format, which may be used directly from your code.

The dataset is relatively small, therefore, like MNIST, you should be able to load the entire thing into memory at once.

2.2 Unpacking

The dataset can be loaded using the Python Pickle module. You can check the Pickle documentation for details on how it works. Furthermore, if you are using Python 2.x, I suggest that you use the cPickle module. This is implemented natively in C, and is therefore much faster. (Python 3.x uses a C implementation by default.)

The loaded Pickle object is a tuple with four elements, which are, in order:

1. The training images.
2. The training labels.
3. The testing images.
4. The testing labels.

The data can, for instance, be loaded and extracted into these four components using the following code:

```
data_file = file("/cifar_data_tf_train_test.pkl", "rb")
train_x, train_y, test_x, test_y = pickle.load(data_file)
data_file.close()
```

A quick note about Python 3: There is, at the moment, a compatibility issue between Pickle in Python 2 and 3. Specifically, when you try to load the CIFAR-10 data, you will get an error like the following:

```
UnicodeDecodeError:
        'ascii' codec can't decode byte 0x89 in position 24:
        ordinal not in range(128)
```

Basically, this happens because Pickle tries to guess the encoding of the data and guesses it wrong. In order to fix it, you have to explicitly specify the encoding to use:

```
train_x, train_y, test_x, test_y = pickle.load(data_file,
                                               encoding="latin1")
```

This should resolve the problem and allow you to load the data correctly.

2.2.1 Training Images

The training images are provided as a four dimensional tensor. The dimensions are, in order:

1. Number of images.
2. Number of image rows.
3. Number of image columns.
4. Number of image channels.

In this case, it yields a tensor of shape $(50000, 32, 32, 3)$.

You can test that your data is valid by using OpenCV to display input images directly:

```
import cv2
...
cv2.imshow("test", train_x[0])
```

Similarly, you can use the same strategy to check the testing data.

2.2.2 Training Labels

These are a straight list of integers between 0 and 9. As you might guess, each integer in this list is the class label for the image at the same index in the training data. No modification should be necessary for the labels.

2.2.3 Testing Data & Labels

These are in the same format as the training data and labels.

2.3 Preprocessing

There are a few steps that need to be taken prior to using the data. The exact implementation of these steps is deliberately left as an exercise to the reader.

2.3.1 Converting to Floats

The data is provided as Numpy arrays of uint8s. It will have to be converted to float32 format before it can be used.

2.3.2 Scaling

For best results, the data should be divided by 255 to scale it between zero and one.

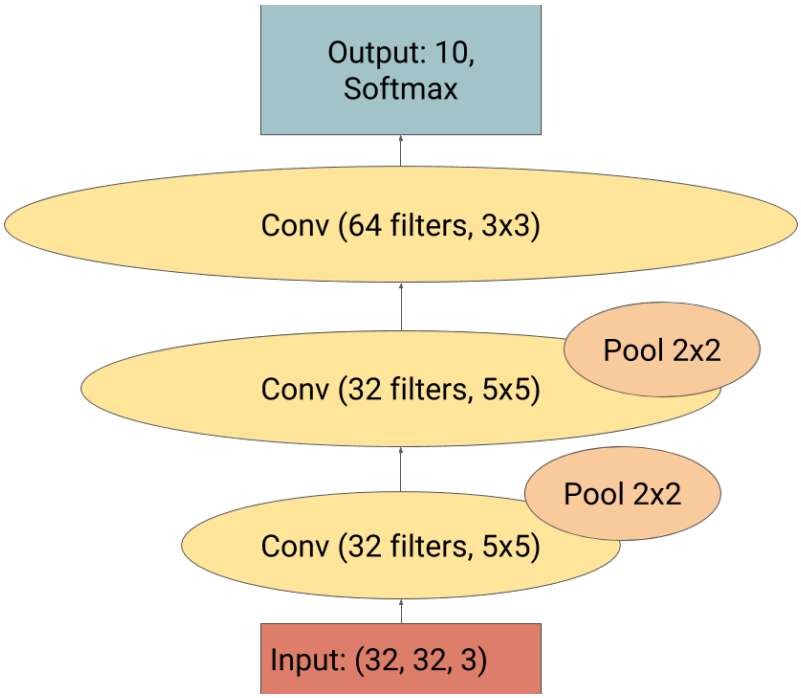

Figure 1: The model that we will be using for this exercise.

2.3.3 Normalizing

The data should be normalized by calculating and subtracting the mean.

3 Implementing the Model

3.1 Model Structure

The model that we will be using is a CNN with three convolutional layers, and two pooling layers. The output is fed into a standard 10-way softmax in order to preform the classification task. Its exact architecture is shown in fig. 1. Every convolutional layer has a stride of 1, and every pooling layer has a stride of 2. No padding should be necessary for any layer.

One thing to note is that *there is no pooling layer after the third convolution.* This is intentional. It is mostly because the output would be too small if we did that.

3.2 Basic TensorFlow Implementation

TensorFlow natively provides most of the functions necessary to implement this model, including conv2d and pool. For the softmax layer, I recommend that

you use sparse_softmax_cross_entropy_with_logits, as that will allow you to use the labels directly without converting to 1-hot format.

3.3 Initialization

For my own work, I used Xavier initialization. (Note that there is a special version for convolutional layers.) This is meant to work reliably on large networks, but works fine on small ones as well, despite not being strictly necessary. If you want to, you are welcome to experiment with other methods of initialization.

The issue that Xavier is trying to solve is that, with large networks that have lots of layers, repeated multiplications and additions tend to compound over time and you end up with exploding or vanishing gradients. This is bad for SGD, because it means that weights with small gradients will effectively not be updated, and weights with large gradients will swing so wildly that, in the worst case, the entire training procedure will be rendered unstable. Theoretically, this can be solved by intelligent weight initialization, which is where Xavier comes in.

Basically, Xavier initialization treats the output to each neuron as a normal distribution with a zero mean. There is some nice math that you can do with normal distributions that essentially says that if your weights are normally distributed, your output will be too, assuming a linear activation.

In Xavier, all our initial weights are drawn from a normal distribution with a mean of zero, and a variance given by:

$$\frac{1}{n_{in}} \tag{1}$$

where n_{in} is the number of inputs to that particular neuron.

In a nutshell, this procedure tends to keep gradient explosion/disappearance in check by ensuring that, statistically, the input to any given neuron should be pretty close to the output in the initial network. This results in gradients that don't vanish or explode, and SGD can therefore touch every parameter without becoming unstable. If you are interested in the nitty-gritty details, I'd recommend this blog post.

3.4 Training

Which optimizer you use to train the model is up to you. I used RMSProp for my own implementation, but anything should work, assuming you choose reasonable hyperparameter values.

In the training run that is detailed in fig. 2, I used a batch size of 500, because I was running on a GPU, and wished to utilize as much of the computational resources as possible. For those of you running on the CPU, however, I would recommend a much smaller batch size, such as 64 or 128. With a smaller batch size, of course, you will likely have to train for a larger number of iterations than I did.

It is also imperative that you decrease the learning rate regularly as you train, as this will allow you to reach the target accuracy. Deciding the exact schedule is left up to you.

Furthermore, the model detailed in this document has been deliberately simplified as much as possible. This should ensure that, even in a CPU-only environment, training should take no more than a few hours.

3.5 Saving

3.5.1 Preparing Your Model

In order for us to validate your model, it has to be in a certain format. Specifically, we need access to particular operations in your TensorFlow graph. In order to facilitate this, we ask that you use a Graph Collection in order to store the following nodes:

- Your placeholder for input data. Assume that the input data has already been pre-processed, as described in the previous section.
- Your prediction operation. This operation should return a row-vector of integers, where each integer is the predicted label for the image at the same index in the input batch.

These nodes should be stored in a collection called "validation_nodes". The following code demonstrates how to do that.

```
inputs = tf.placeholder(...)
predict_op = ...

...

# Create the collection.
tf.get_collection("validation_nodes")
# Add stuff to the collection.
tf.add_to_collection("validation_nodes", inputs)
tf.add_to_collection("validation_nodes", predict_op)
```

3.5.2 Exporting Your Model

When you are done training, you will need to save your model in a form that allows us to grade it. For this, we ask that you use the built-in TensorFlow Saver class. You can save your model using the following code:

```
saver = tf.train.Saver()

with tf.Session() as session:
    ... Train the model ...
    save_path = saver.save(session, "my_model")
```

(The "validation_nodes" collection that you created will automatically be exported.)

There are going to be two important files that this will generate. One is the .meta file, and the other will be a checkpoint file. (With a .data extension.) The former details your actual model structure, and the second contains the saved weights. You will need to send us **both** of these files for us to grade your model.

4 Expected Results

4.1 Validation Performance

The final model should achieve a little over 70% accuracy on the testing set. Since we are foregoing data augmentation for the sake of simplicity, expect some mild overfitting, but nothing crazy. Fig. 2 shows a sample training graph. Note that this graph was made with a batch size of 2500, so 1 iteration means that 2500 images were processed. Your results need not look exactly like this, but they should be broadly similar. During the grading process, your model will be evaluated on the 5000 images that were held out from the testing set, so your "official" results may differ slightly from your own experiments.

4.2 Visualizing Filters

Part of the assignment is to visualize the filters in the first layer. An example output is shown in fig. 3.

In order to do this, you will have to run gradient ascent in order to maximize the activations of a particular filter. A good example of how to do this can be found here. In this case, is enough to follow the procedure in the "naive feature visualization" section, there is no need to generate images on multiple scales or anything like that.

5 Final Notes

Hopefully, this is enough information to get started with CNNs and the CIFAR-10 dataset. If you have questions, or you think that this guide has a mistake or should include something that it doesn't, please email me at pettid@rpi.edu.

6 Appendices

List of Figures

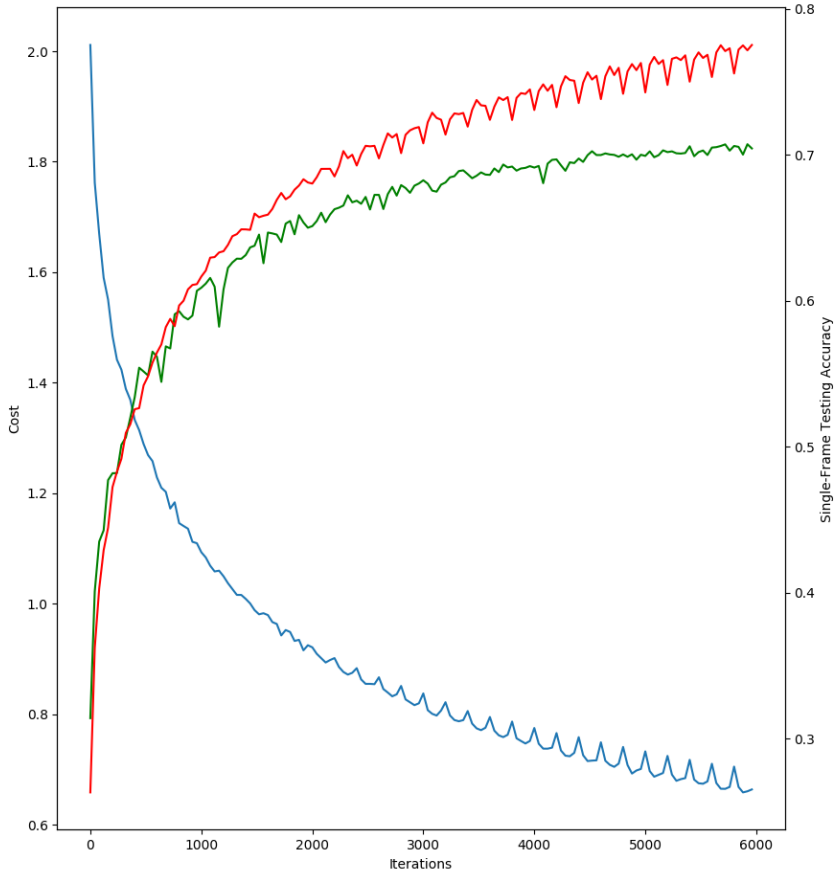

Figure 2: A sample training graph for this model. The red line is training accuracy, the green line is testing accuracy, and the blue line is cost.

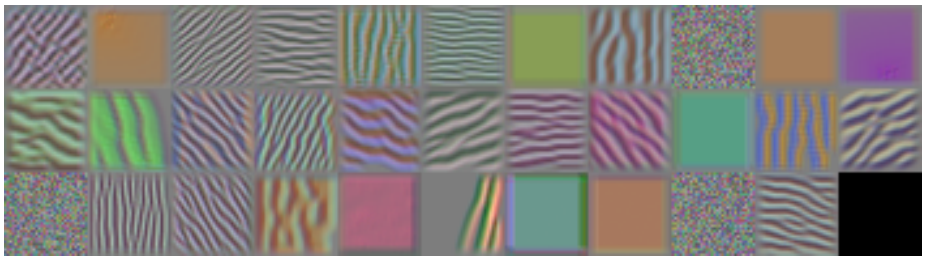

Figure 3: A sample visualization of all the filters in the first layer.