Table of Contents

[1. Objective 2](#_Toc472195896)

[2. Project details 2](#_Toc472195897)

[3. Amazon setup 2](#_Toc472195898)

[4. Rubric 2](#_Toc472195899)

[5. Data Set 3](#_Toc472195900)

[5.1. Training and testing data set 4](#_Toc472195901)

[5.2. New images for sanity testing 4](#_Toc472195902)

[6. Dataset Exploration 4](#_Toc472195903)

[6.1. Loading data 4](#_Toc472195904)

[6.2. Data Summary 5](#_Toc472195905)

[6.3. Explore Data 5](#_Toc472195906)

[7. Design 5](#_Toc472195907)

[7.1. Preprocessing 6](#_Toc472195908)

[7.2. Normalize the input pixel 6](#_Toc472195909)

[7.3. Model Architecture 8](#_Toc472195910)

[7.4. CNN model Option 9](#_Toc472195911)

[7.5. CNN model Option 11](#_Toc472195912)

[7.6. CNN model Option 11](#_Toc472195913)

[7.7. CNN model Option 11](#_Toc472195914)

[7.8. CNN model Option 11](#_Toc472195915)

[7.9. CNN model Option 11](#_Toc472195916)

[7.10. Training 12](#_Toc472195917)

[7.11. Measuring Accuracy 14](#_Toc472195918)

[8. Putting together 14](#_Toc472195919)

[9. Programming Tips 15](#_Toc472195920)

[9.1. Image size reduction 15](#_Toc472195921)

[10. Testing and Optimization 15](#_Toc472195922)

[10.1. Low accuracy 16](#_Toc472195923)

[11. Further Research and Ideas 17](#_Toc472195924)

[12. Conclusion 17](#_Toc472195925)

# Objective

This is an article to provide my thoughts on an interesting project I did for the Udacity Self-Driving Car Nanodegree.

The code and technical details can be found ….

The goal is to teach a Convolutional Neural Network (CNN) to classify tariff sign iamges

# Project details

1 In this project, you will use what you've learned about deep neural networks and convolutional neural networks to classify traffic signs. Specifically, you'll train a model to classify traffic signs from the [**German Traffic Sign Dataset**](http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset).

After you train your model, you will test it on new images of traffic signs you find on the web. Or, if you're feeling adventurous, test your model on pictures of traffic signs you find locally!

# Amazon setup

1. [**Follow the Udacity instructions**](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/614d4728-0fad-4c9d-a6c3-23227aef8f66/concepts/f6fccba8-0009-4d05-9356-fae428b6efb4) to launch an EC2 GPU instance with the udacity-carnd AMI.
2. Complete the **Setup** instructions.
3. Install TensorFlow with GPU support. All of the necessary dependencies are pre-installed on the AMI, so you can jump straight to installing the package: pip install tensorflow-gpu

# Rubric

Dataset Exploration

|  |  |
| --- | --- |
| Dataset Summary | Student performs basic data summary. |
| Exploratory Visualization | Student performs an exploratory visualization on the dataset. |

Design and Test a Model Architecture

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Preprocessing | Students provides sufficient details of the preprocessing techniques used. Additionally, the student discusses why the techniques were chosen. |
| Model Architecture | Student provides sufficient details of the characteristics and qualities of the architecture, such as the type of model used, the number of layers, the size of each layer. Visualizations emphasizing particular qualities of the architecture are encouraged. |
| Dataset and Training | Student describes how the model was trained and evaluated. If the student generated additional data they discuss their process and reasoning. Additionally, the student discusses the difference between the new dataset with additional data, and the original dataset. |
| Solution Design | Student thoroughly discusses the approach taken for deriving and designing a model architecture fit for solving the problem given. |

Test a Model on New Images

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Acquiring New Images | Student chooses five candidate images of traffic signs taken and visualizes them in the report. Discussion is made as to any particular qualities of the images or traffic signs in the images that may be of interest, such as whether they would be difficult for the model to classify. |
| Performance on New Images | Student documents the performance of the model when tested on the captured images and compares it to the results of testing on the dataset. |
| Model Certainty Visualization | The softmax probabilities of the predictions on the captured images are visualized. The student discusses how certain or uncertain the model is of its predictions. |

# Data Set

## Training and testing data set

## New images for sanity testing

The images don't really need to be labeled because you will know what the correct classification is for each image, you just have to see what the model predicts it to be and compare that to what you know the true classification of the image to be. If you want them to be labeled you can just add labels to them manually.

 took screenshots from Google StreetView and pulled out the traffic signs. It provided a pretty good variation of signs and environments. I had much more success doing it this way. Google search didn't provide a very good set of results for me.

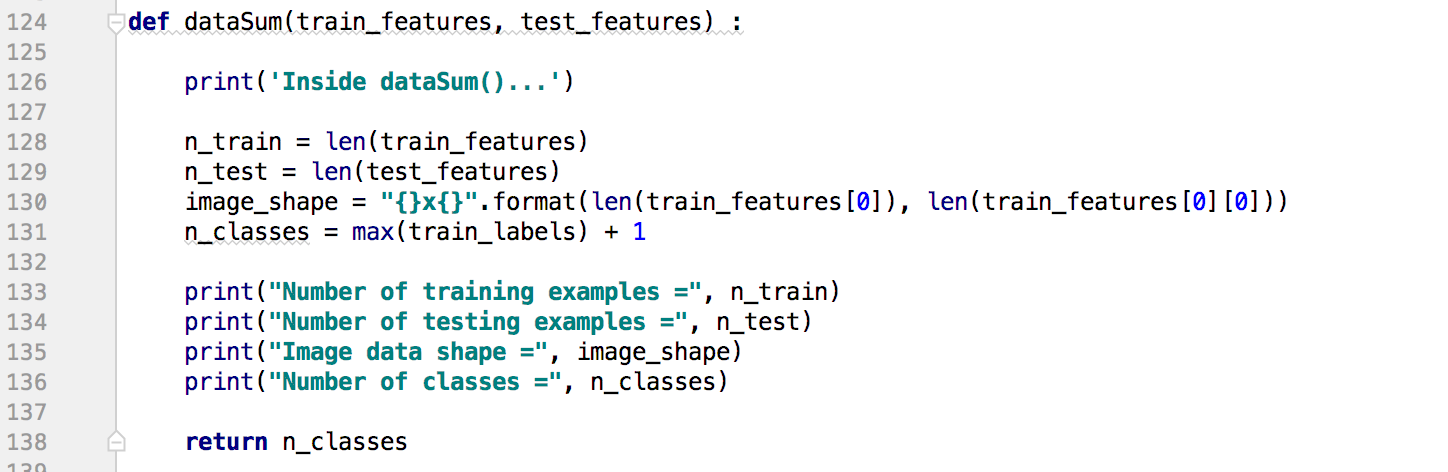
It has to be  Now how do I make it 32x32?

# Dataset Exploration

## Loading data



## Data Summary



## Explore Data



# Design

## Preprocessing

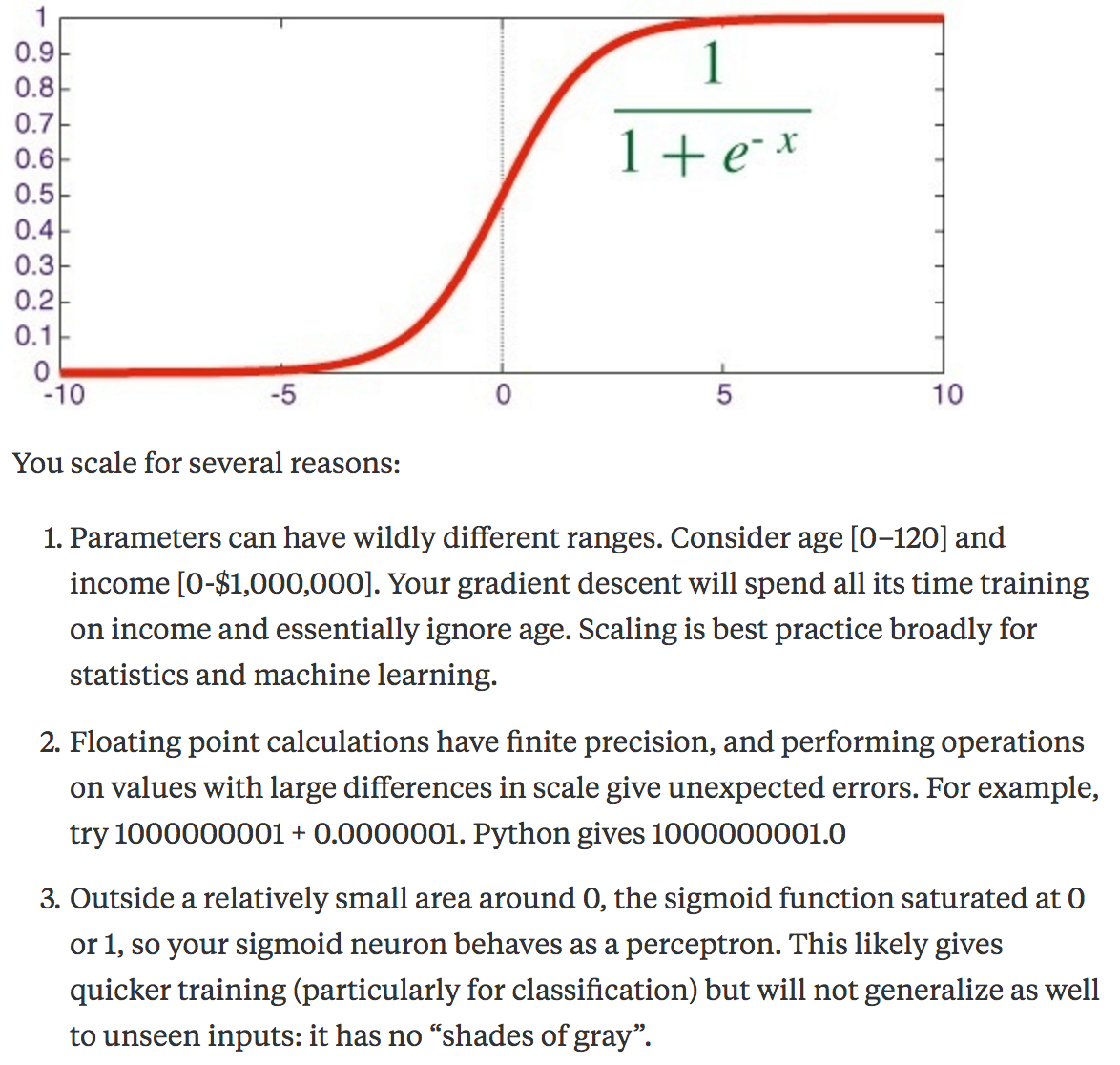
## Normalize the input pixel

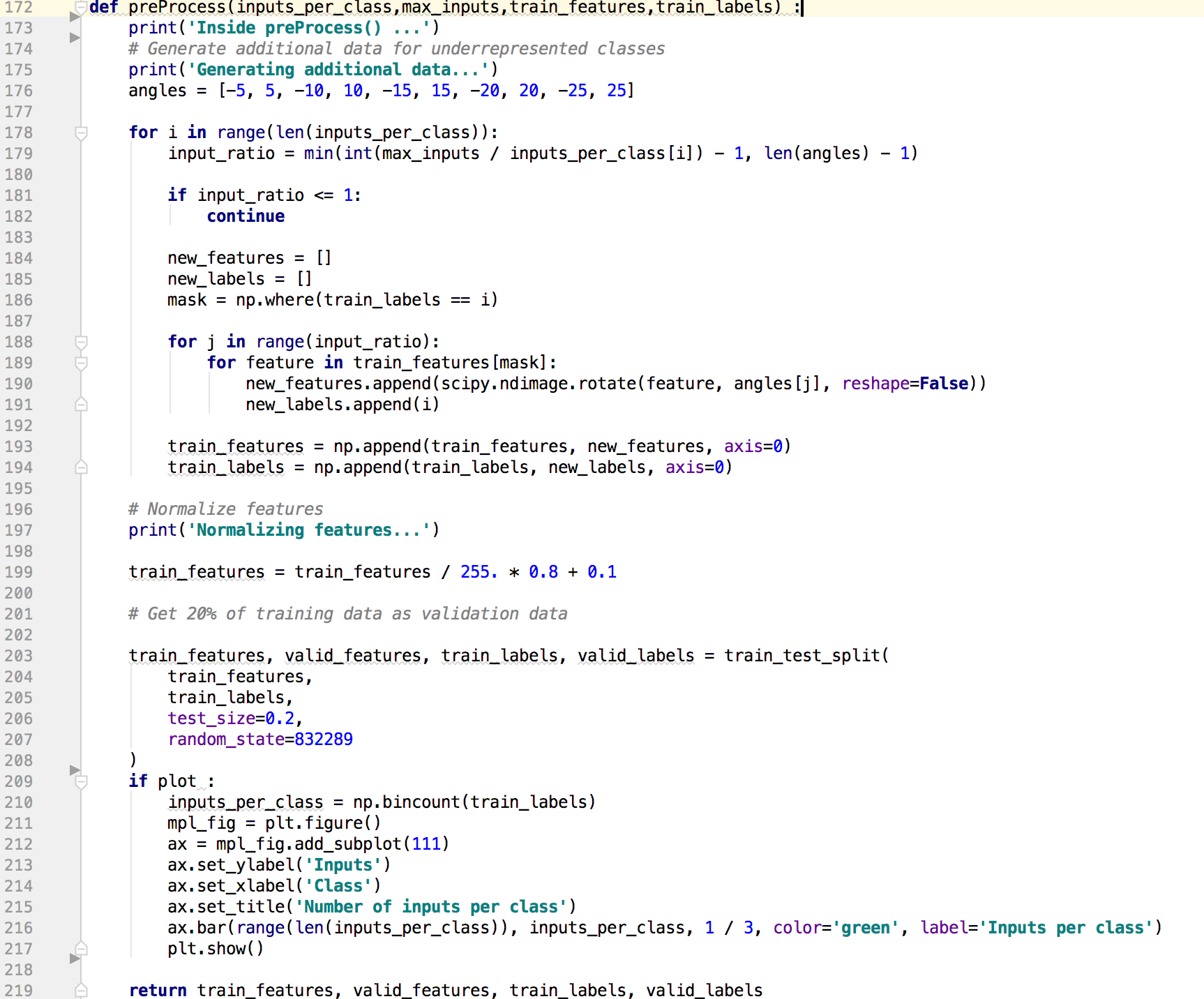
In Neural Network, we learn continually by adding gradient error vector which is **multiplied by learning rate**.

In our training data, if we don’t scale our features, the distributions of feature values would likely be different for each features, and thus the learning rate would cause corrections in each dimension that would differ from one another.

This is non-ideal as we might find ourselves in a oscillating (unable to center onto a better maxima in cost(weights) space) state or in a slow moving (traveling too slow to get to a better maxima) state.

Thus, we try to normalize features before using them as input into NN (or any gradient based) algorithm.



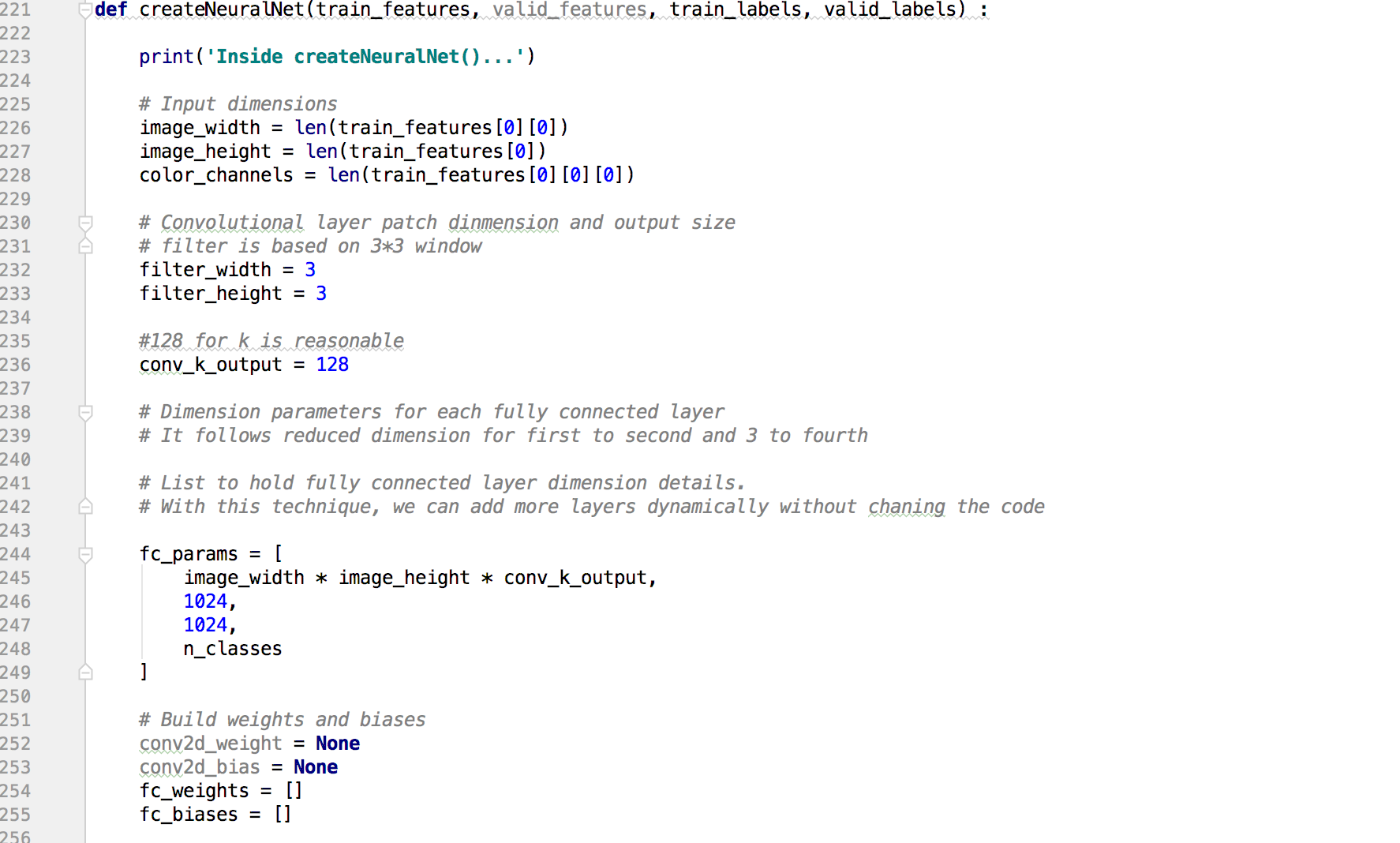


## Model Architecture

The CNN I chose is a pretty standard CNN consisting of 4 convolutional layers with ReLU activations, followed by two fully connected layers with dropout regularization.

Finally a single neuron formed the output that predicted the steering angle. One noteworthy thing is the absence of pooling layers. The rationale behind avoiding pooling layers

I trained the network on an Ubuntu 16.04 system using an NVIDIA GTX 1080 GPU. For any given set of hyperparameters the loss typically stopped decreasing after a few epochs (200000 images each).



## CNN model Option

# Input images

image\_height = image\_width = 32

color\_channels = 3

x = tf.placeholder(tf.float32, [None, image\_width, image\_height, color\_channels])

# Outputs

n\_classes = 43

y = tf.placeholder(tf.float32, [None, n\_classes])

logits = model(x, weights, biases)

batch\_size = 128

training\_epochs = 30

layer\_width = {

'layer\_1' : 32,

'layer\_2' : 64,

'layer\_3' : 128,

'fully\_connected' : 512

}

weights = {

'layer\_1' : tf.Variable(tf.truncated\_normal([5, 5, 3, layer\_width['layer\_1']])),

'layer\_2' : tf.Variable(tf.truncated\_normal([5, 5, layer\_width['layer\_1'], layer\_width['layer\_2']])),

'layer\_3' : tf.Variable(tf.truncated\_normal([5, 5, layer\_width['layer\_2'], layer\_width['layer\_3']])),

'fully\_connected' : tf.Variable(tf.truncated\_normal([1024, layer\_width['fully\_connected']])),

'out' : tf.Variable(tf.truncated\_normal([layer\_width['fully\_connected'], n\_classes]))

}

def model(inputs, weights, biases):

# Convolute and max pool inputs to create first layer

layer1 = conv\_n\_rect(inputs, weights['layer\_1'], biases['layer\_1'])

layer1 = max\_pool(layer1, filter\_size=2)

layer2 = conv\_n\_rect(layer1, weights['layer\_2'], biases['layer\_2'])

layer2 = max\_pool(layer2, filter\_size=2)

layer3 = conv\_n\_rect(layer2, weights['layer\_3'], biases['layer\_3'])

layer3 = max\_pool(layer3, filter\_size=2)

# Apply fully connected layer to previous layer

fully\_connected\_layer = tf.reshape(layer3, [-1, weights['fully\_connected'].get\_shape().as\_list()[0]])

fully\_connected\_layer = tf.add(tf.matmul(fully\_connected\_layer, weights['fully\_connected']), biases['fully\_connected'])

fully\_connected\_layer = tf.nn.relu(fully\_connected\_layer)

# Create final prediction layer

model = tf.add(tf.matmul(fully\_connected\_layer, weights['out']), biases['out'])

return model

def conv\_n\_rect(inputs, filter, biases, strides=1):

# Convolute input with filter

layer = tf.nn.conv2d(inputs, filter, strides=[1,strides,strides,1], padding='SAME')

# Add bias

layer = tf.nn.bias\_add(layer, biases)

# Apply activation function

return tf.nn.relu(layer)

def max\_pool(layer, filter\_size=2):

return tf.nn.max\_pool(layer, ksize=[1,filter\_size,filter\_size,1], strides=[1,filter\_size,filter\_size,1], padding='SAME')

logits = model(x, weights, biases)

# Determine the loss value

cost = loss(logits, y)

# Define the optimizer

optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate).minimize(cost)

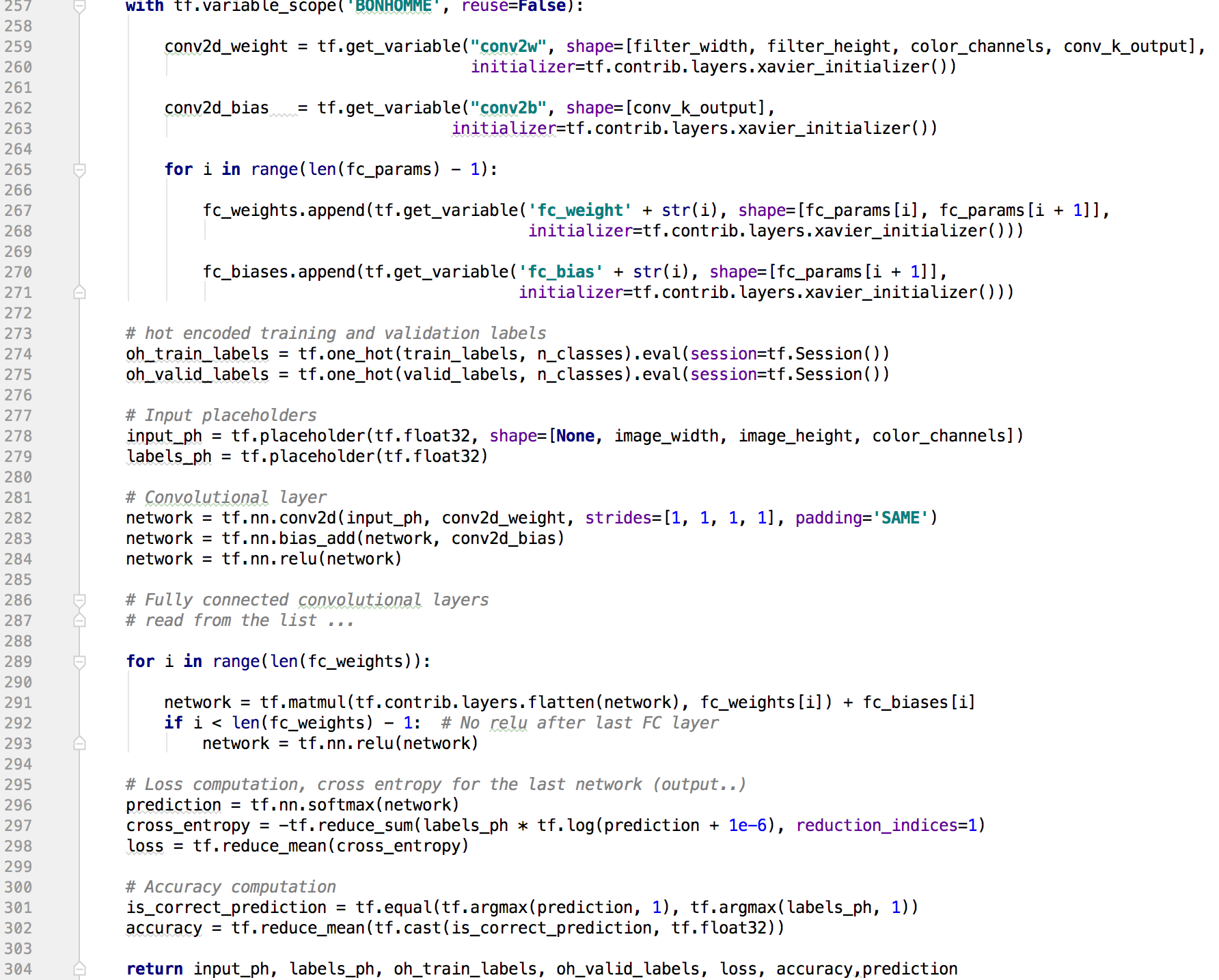
## CNN model Option

## CNN model Option

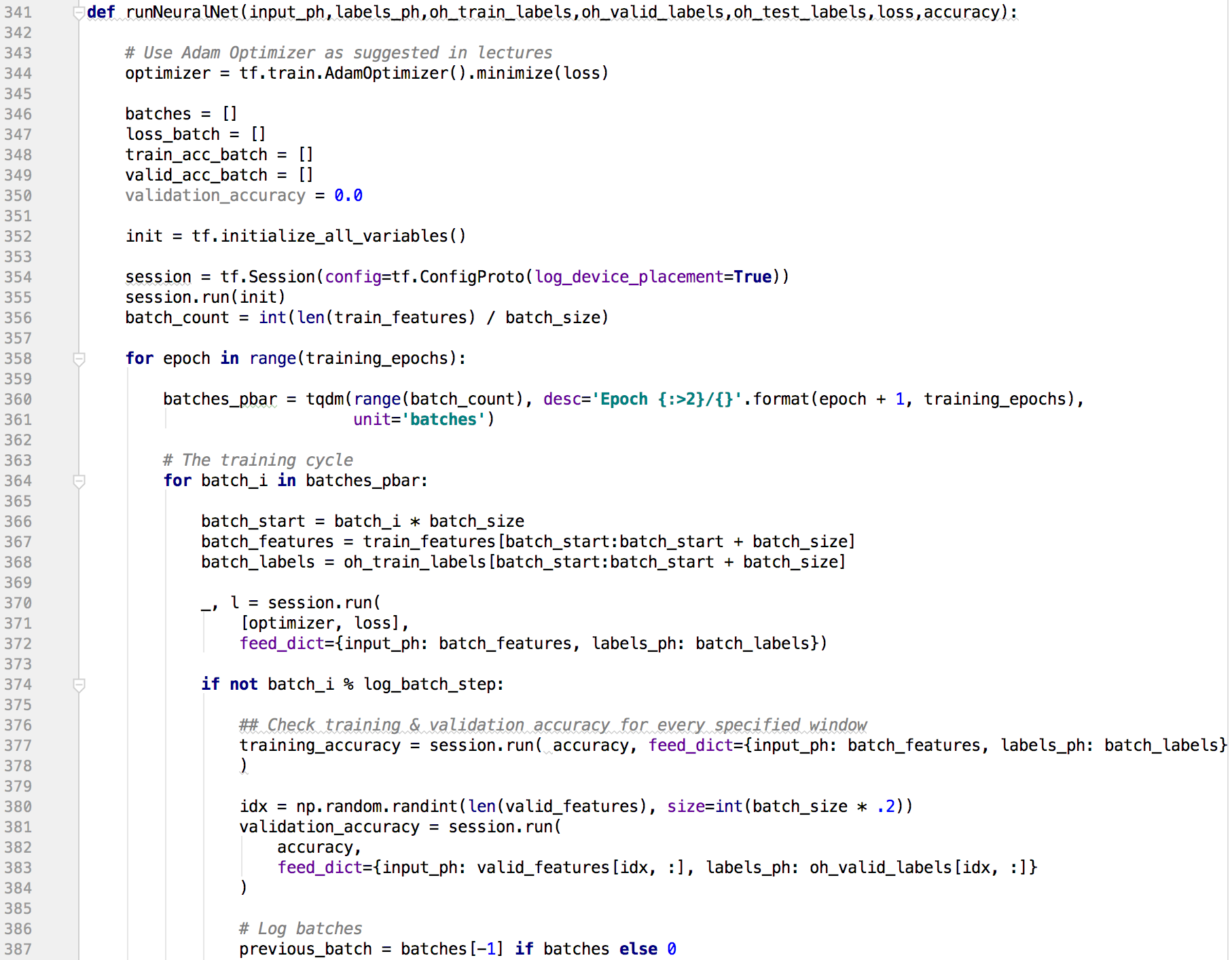
## CNN model Option

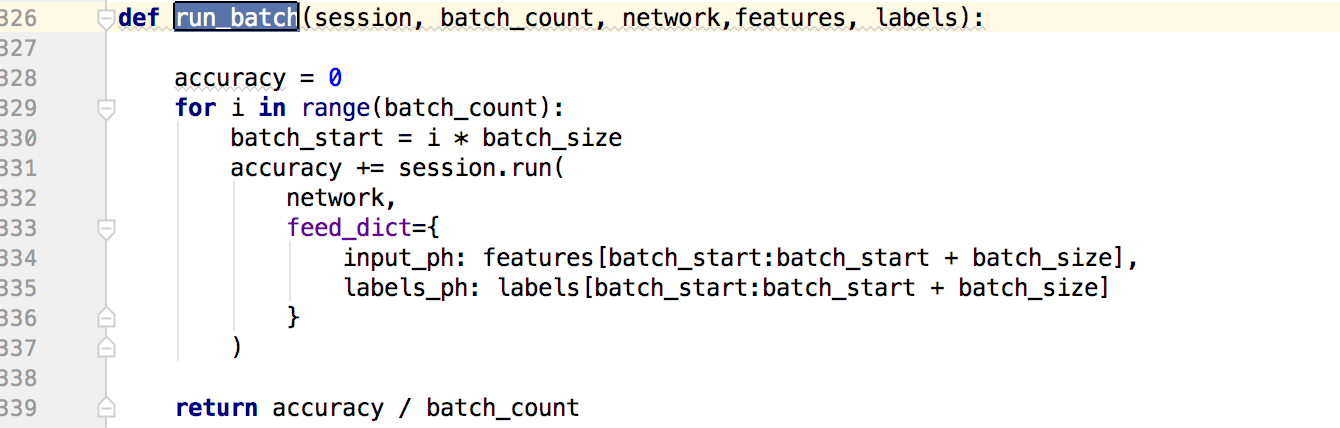
## CNN model Option

## CNN model Option

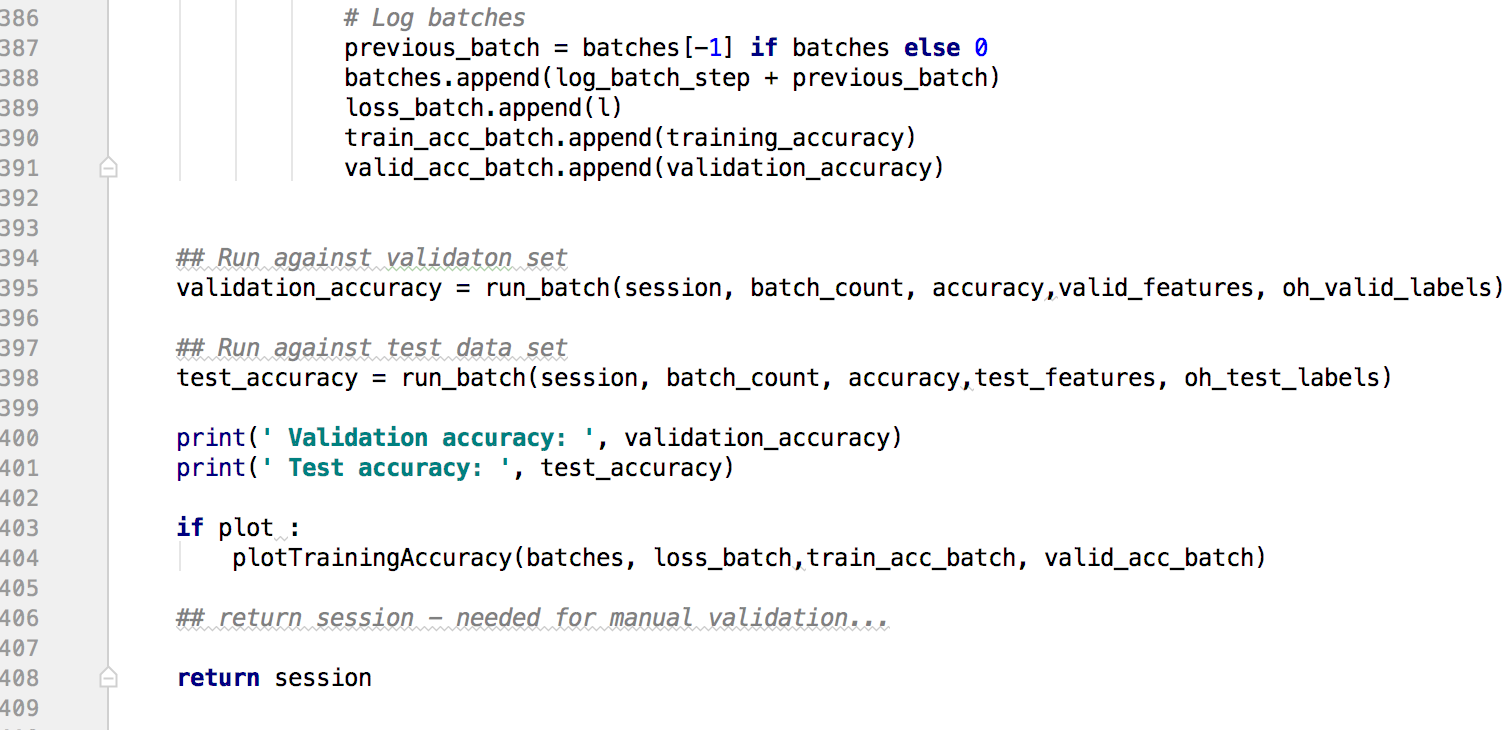


## Training





## Measuring Accuracy



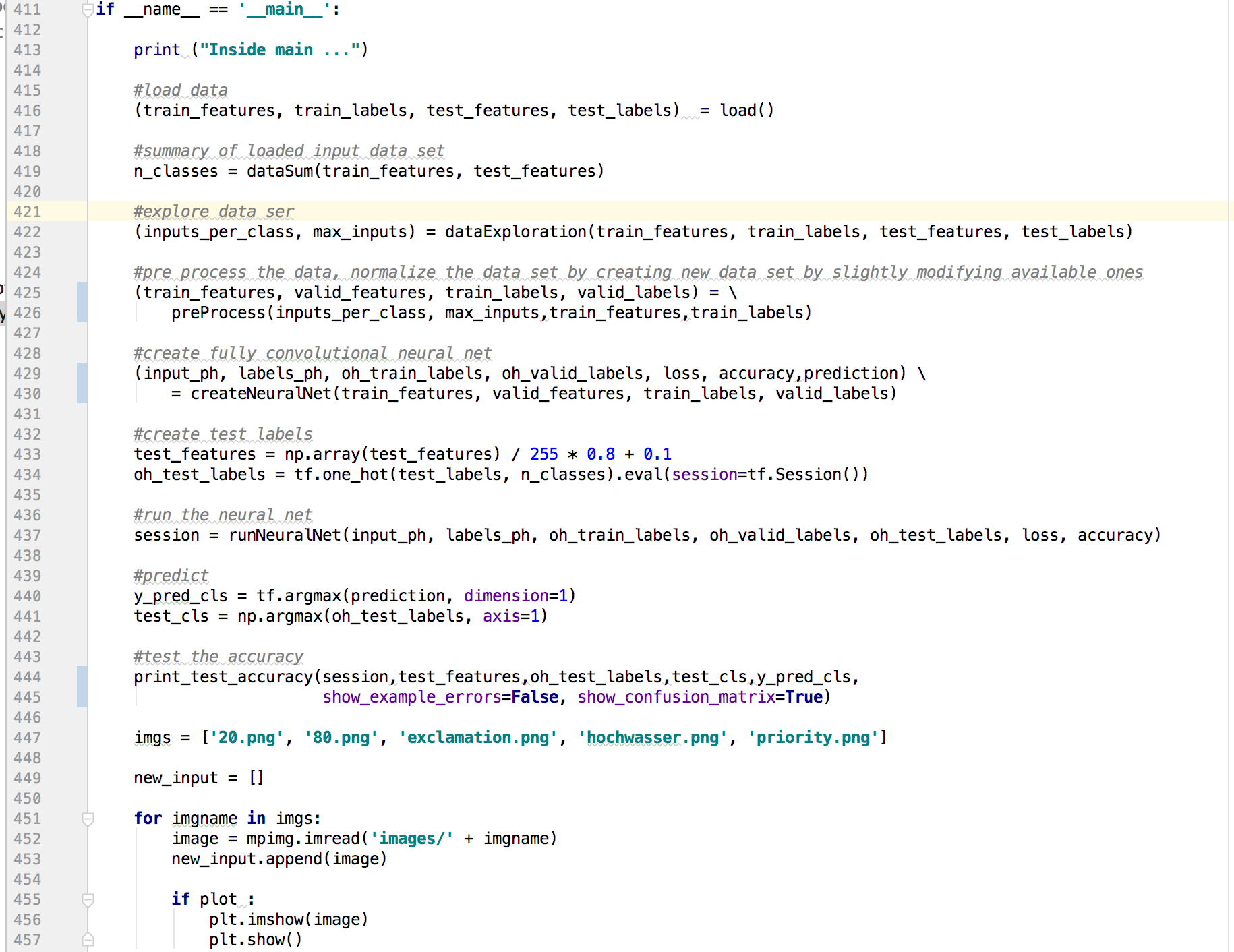
In most DL libraries such as Keras, training accuracy is calculated by concatenating the outputs of the forward propagations during training (the training data needs to be passed through the network in order to calculate the gradients, so we just take the predictions of the network at that point). The benefit of this is that it gives a rough estimate of how the network is training, without requiring extra computation.

However, the downside of this is that is not a accurate demonstration of how the network behaves (since the network is different at the start of the epoch when the first training data is predicted on) - especially during the first epoch where the start has very high loss. In addition, this means that things such as regularization and dropout are preserved, which will increase the perceived loss on the training set. [See more](https://keras.io/getting-started/faq/#why-is-the-training-loss-much-higher-than-the-testing-loss)

If you want an accurate measurement of the training metrics, then your best bet is to predict on the training set at the end of the epoch as if it was another test set, with dropout turned off. This will let you know how well the network actually does on this data (and can let you see how much it is overfitting.)

# Putting together

Following is the flow of traffic sign classifier.



# Programming Tips

## Image size reduction

resized\_image1 = cv2.resize(image1, (32, 32))

# Testing and Optimization

Experimentation is the best way to design a good architecture and find optimum parameters. And I wont call it trial and error as there is already lot of research done on the topic. Here are some of the techniques and papers that I used -

Paper1- [Systematic evaluation of CNN advances on the ImageNet](https://arxiv.org/abs/1606.02228)

Summary

1. Preprocessing- create a two layered network to learn color space.
2. Max pooling combined with avg pooling gives better accuracy.
3. [Batch normalization](https://arxiv.org/abs/1502.03167) + Relu is better.
4. If not Batch Norm then use ELU
5. Implement [L2 regularization](https://arxiv.org/abs/1206.5533)   
   And many other topics like, batch size, data size, bias, etc.

Paper2 - [Dropout: A Simple Way to Prevent Neural Networks from Overfitting](https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf)

Summary - Use dropout value of 0.5 while training

Paper3&4 - [Going Deeper with Convolutions](https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf) & [Deep Residual Learning for Image Recognition](https://arxiv.org/pdf/1512.03385v1.pdf)

Summary - Decreasing the computation complexity by using bottle next layers.

Howevery this is just scratching the surface, they are lot of resources to play around with big grin)

Have a look at the CS231n [lecture videos](https://www.youtube.com/playlist?list=PLLvH2FwAQhnpj1WEB-jHmPuUeQ8mX-XXG), particularly videos 5, 6 and 7. Lots of advice on how to approach training, parameter tuning, how to deal with overfitting etc. There will always be an element of trial and error though.

## Low accuracy

The issue was that the training error was low. To diagnose this issue, I lowered the learning rate, took a much smaller network that can be overfit easily (one of my tricks to diagnose neural networks code), and ran the training routine. What I noticed is that the accuracy was first increasing, and then it suddently went down to below 1%. I then checked values of class labels, and all were returned as same. This indicated that there must be mistake in softmax calculation. After some googling and stackoverflowing, found the reason to be ill-conditioned values being given to log calculations. I therefore changed the script of log calculation by truncating value between 1e-10 and 1, and that fixed the issue. Below is code snipped and example of intermediate accuracies before and after clipping y\_conv value.

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(tf.clip\_by\_value(y\_conv,1e-10,1.0)),

reduction\_indices=[1]))

https://carnd-forums.udacity.com/questions/12619143/one-reason-for-low-accuracy-ill-conditioned-value-for-log-calculation

# Further Research and Ideas

# Conclusion

Summarizing, this was a really interesting project. It would be interesting to see whether recovery events can also be simulated from real world data. Currently, I can’t see why not. The project cost me countless of hours of sleep over a two week period of time, gray hairs and cursing included, but the result was well worth it. Deep learning is an exciting field and we’re lucky to live in these times of discovery.

For more details please check out the code and the readme here:  
<https://github.com/ksakmann/CarND-BehavioralCloning>

Template statements

* The results came as a pleasant surprise after several nights without any progress.
* I was surprised how well the …
* The performance on the training track was a little

<https://www.quora.com/On-what-kinds-of-data-besides-natural-images-do-convolutional-neural-networks-shine>

Various Implementations

<https://github.com/garyl2203/traffic_signs_P2/blob/master/Traffic_Signs_Recognition_Ver5.ipynb>

https://github.com/jmlb/Udacity-SDCND/blob/master/CarND-TrafficSigns-P2/Traffic\_Signs\_Recognition.ipynb