Project 1

Neural Net Hyper Parameter Optimization

Alex Harper

Table of Contents

Illustration Index

Introduction

For another class, I was to make a neural net to try to do classification of data. After putting the net together, and testing it, I was not sure that my solution was optimal for several different measures; the number of the layers, the rate of decreasing size, the width of the layers, the dropout rate for the training step. For this project, I will attempt to make sense of the effect of different parameters on the training time and the level that it's accuracy plateaus at.

What is a Neural Net

A neural net is a computation graph, where the different nodes are called neurons and each neuron does a single simple calculation. The neurons are grouped into layers, and the different layers are connected to each other. The connections are called weights and are simply a scaling value. Data comes into the layer, is multiplied by the weights, and then run through the activation function of the neurons. The results in an output that moves on to the next layer to have the same set of operations applied. Below is an example, where the circles are the neurons with their activation function, and the arrows represent the weights. Notice how a weight connects every neuron to every other neuron in the layer before and after it.

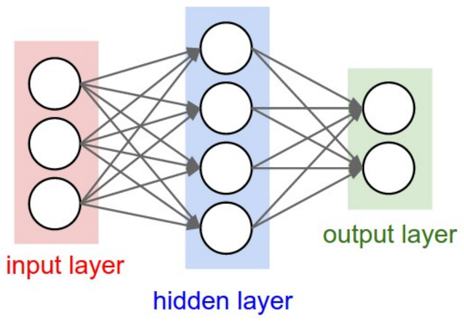

Illustration 1: A simple neural net example

Training a neural net lets it learn what it is supposed to do to process the input data into correct output data. This training changes the weights between the layers, which change the output of the connected neurons, and the neurons later in the net. How this change is determined is beyond the scope of this paper, but it is import to know that the bigger the layers, and the more layers there are, the more weights there are to change for training, and hence take more time. The reason you want more layers though, is to have the network learn for more complex ideas, where the more layers you have, the more capable of understanding a complex problem.

What Are We Looking At

The structure of the net the I made is using the idea of the "wide and deep" architecture. The idea is that sometimes the answer only needs a small number of complex concepts to mix in with some simple concepts. The architecture has two parts of the deep side that computes more complex concepts and then a wide side that computes lots of simple concepts. These two sides are then combined for processing in the output layer.

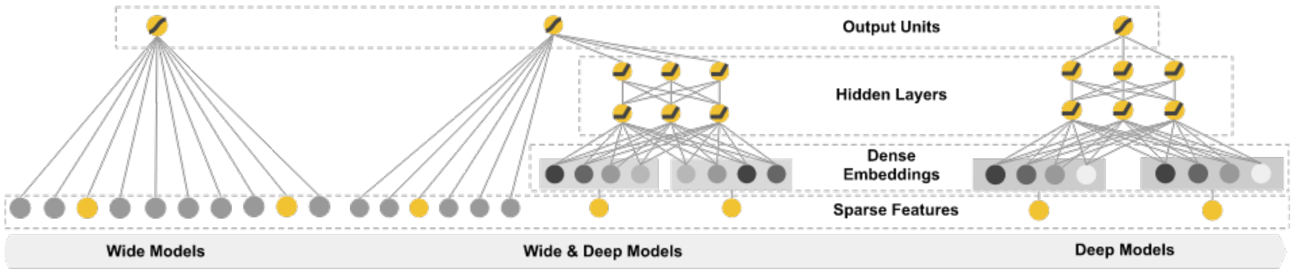

Illustration 2: The Wide and Deep Architecture

The different parameters that we are looking at adjusting are the number of the layers, the rate of decreasing size, the width of the layers, and the dropout rate for the training step.

- Number of layers – how many of the hidden layers there are in the net, highlighted in blue in illustration 1
- Rate of decreasing size – not any official term but referring to the size of each hidden layer in relation to the one before it. If we have 3 hidden layers and a rate of .4, then it might be something like [layer0 = 10, layer1=4, layer2=1] using the formula $size = starting\_size * rate^{depth}$
- Width of the wide layer – how many neurons are in the shallow layer
- Starting Width of the deep layers – the number of neurons in the first layer (layer0) of the deep side of the network
- Dropout rate – the percentage of the neurons that are artificially set to 0 during the training step

What we are trying to optimize to some degree is both the time it takes to train and the accuracy that the network can achieve. The more important measure is the accuracy as that is the reason for making the network. The time to train is more of making a compromise with accuracy where taking a month longer to train for .001% better is usually not economical. At the end, it is a more intuitive decision to be made, but needs a bit of information to decide, and graphs help a lot.

The Data

Knowing what we are trying to optimize, next comes looking at the data. I left my computer running for 14 hours to get this data; the GPU being bottle necked by the CPU. All the data was saved in a CSV file so that it can easily be import by the pandas package. After importing, it was graphed in different ways to try to find the optimized parameters for the network.

Before we start looking at individual parameters, lets see how much variance there is in the data. If there is nearly no change in the way it behaves with all of the options, then it would be best to simply take the simplest set and move on. But as we can see in the graph below, there is a difference between the maximum values and the minimum values for all the data. This probably means that some parameters are making a useful difference in how it behaves.

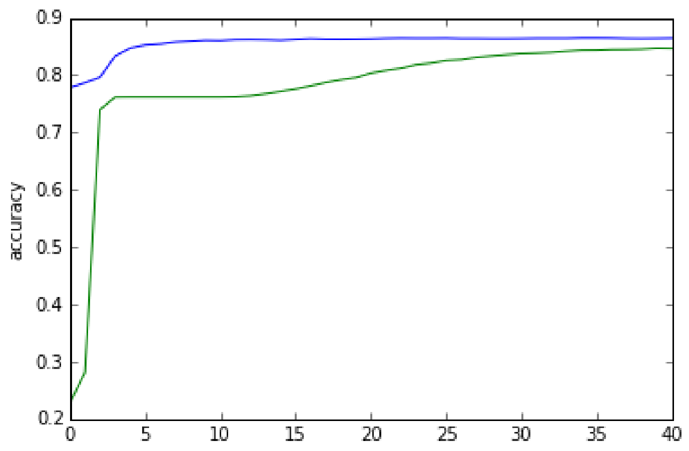

Illustration 3: Comparing the Maximum accuracy to the Minimum accuracy

Number of Layers - Deep

Looking at parameters, lets begin by comparing the max training reached per depth on the deep side of the net. As you can see below, there is no real difference in the final accuracy, but the shallower layers get to the maximum sooner. 6 layers is notably slower than the others to get the last couple percentage, not getting to the max until about 3000 training steps. 3 and 4 deep layers are about the same, reaching the maximum around 2500 training steps, but for now we will choose 3 deep as the only option to take forward.

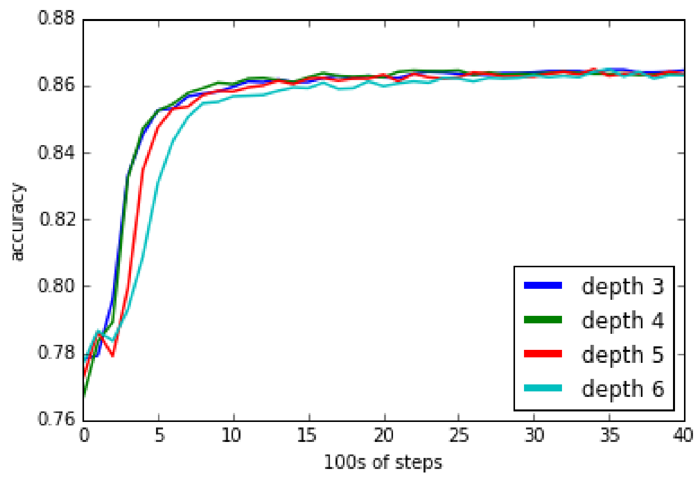

Illustration 4: Comparing the best efforts of the different layer depths

Well, since 3 layers seems deep enough, lets see how much variance it has itself. Interesting thing to note is the oddly bulgy variance at the beginning, but it seems to converge after roughly 1500 training steps in. Also note it gets very small variance between min and max by 2500 steps; I would say that there is no point in training further than that with only 3 layers no matter the other parameters we choose.

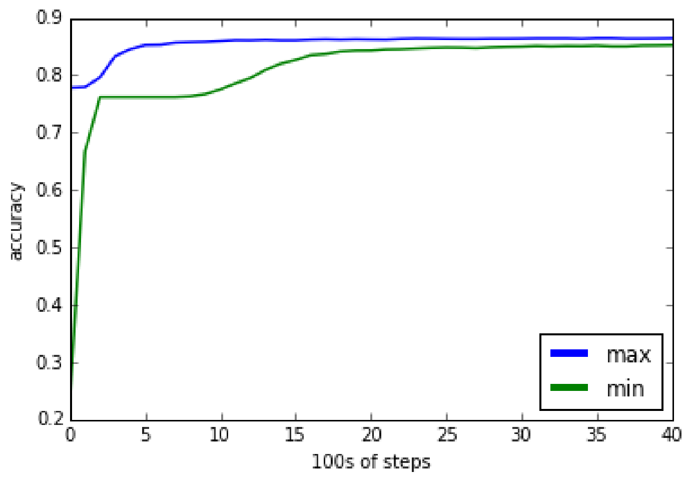

Illustration 5: Looking at the variance of accuracy for all 3 deep options

Lets now look at all the different options at once and see if there is any major divide in the data. If there is, then that means there is likely a single factor that is simply worse in one state. Looking at the graph, besides being a giant rainbow, the lines seem to simply converge to the same maximum with a fairly even spread, even in the bulge on the left. That means there does not seem to be any option that is binary good or bad.

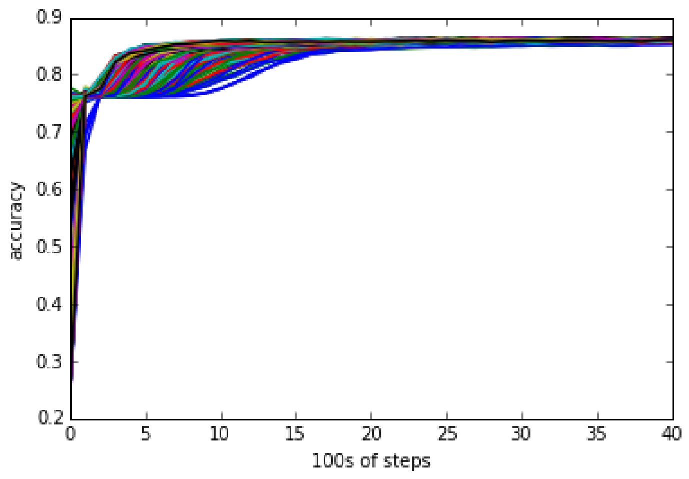

Illustration 6: Viewing all the options accuracy at once for 3 deep

Width of Layers – Deep

Since there is not much difference in the maximum accuracy (likely only varying due to random weight initialization) then the other factor left to optimize is training time. Lets start by taking look at the most obvious thing to affect the time, how wide the starting layer for the deep side of the net is. Interesting to note, is that the largest starting size is the fastest to converge, not really ever having the bulge area affect it.

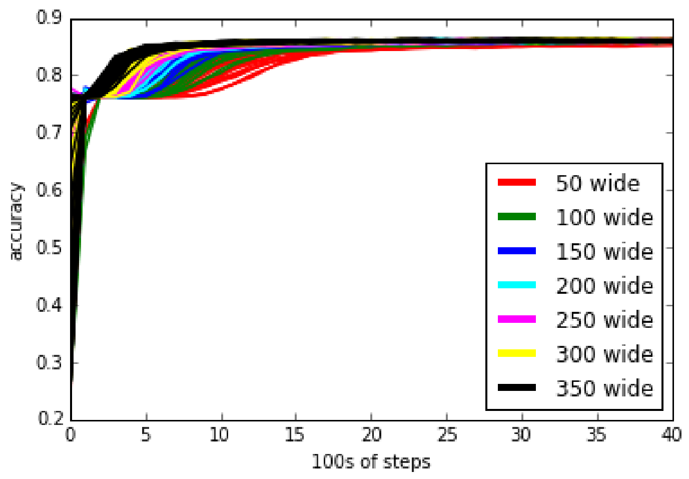

Illustration 7: Comparing different starting sizes for the deep side of the neural net

Though we think we know exactly what is going on, lets trim it down to zoom in on the bulge area to see more details. This graph simply has the x and y axes zoomed in so that we can try to see the groups more clearly. The left graph shows all the groups, which looks a little messy still, but we can see that the red lines are probably the worst off. The red group is the smallest number of starting neurons in the deep part of the network. In the graph to the right of it, it is easy to see that no lines of the red are close to the black, so we can just outright remove that option.

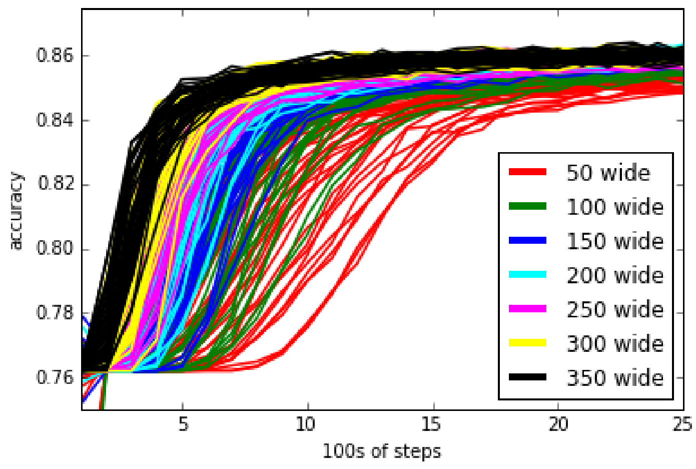

Illustration 8: All starting widths for the deep side

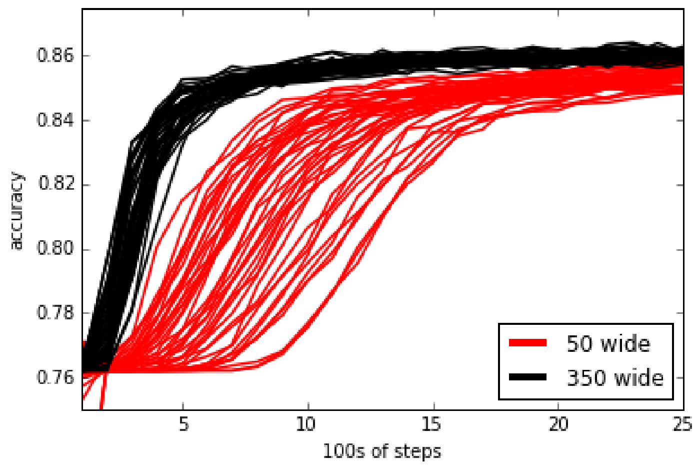

Illustration 9: The largest and smallest starting widths for the deep side

The next two graphs show different groups compared to the black. Blue and green show an improvement over the red, but are in the same situation where they are not close to black, and so we can remove those options also. The yellow and pink lines are touching or covered up by the black lines, but pink is obviously sticking out a bit so we can remove that option, leaving yellow. Yellow seems to be mostly covered up by the black lines, and so it might be best to leave the option until we look at the other parameters. This will help us tease out what other parameters make a difference in training time by comparing how well the options work between these two groups.

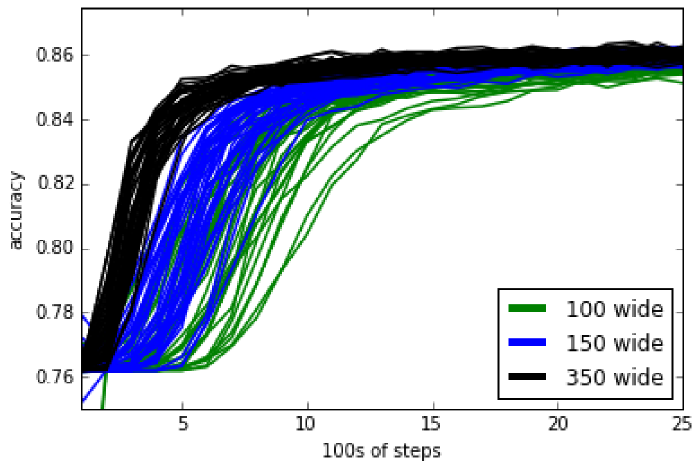

Illustration 10: Two medium starting widths for the deep side

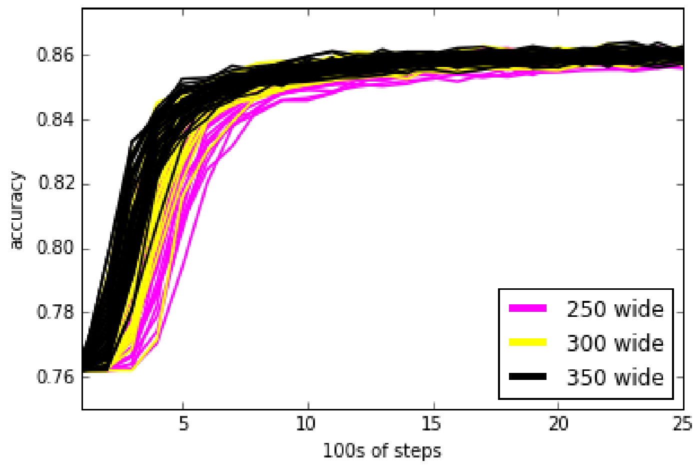

Illustration 11: The largest starting widths for the deep side

The Other Parameters

At this point, we know that we are near the top in how quickly we can train the net. For the most part, the next parameters do not seem to make a big difference in how quickly it trains and really we are nitpicking. But lets take a look anyways, and see if there are any conclusions we can draw.

This first graph is what i am calling the decay rate for the width of the layers in the deep part of the net. Due to time constraints, only two values were tested, but we can see that the green lines seem to be slightly faster to train on average, which corresponds to wider layers in general on the deep side. The red lines are not really much worse, and so really probably doesn't matter much.

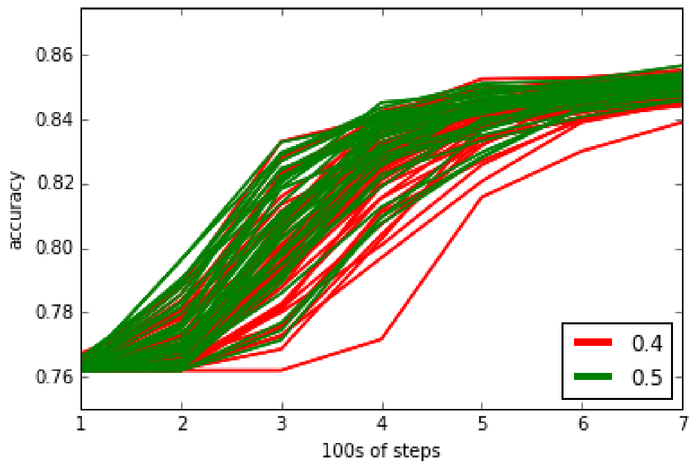

Illustration 12: Decay rates of the layer widths

This next graph shows the 3 different tested dropout rates. We can see that the blue and green lines are mostly in the same area, but the red lines are noticeably slower. The faster rates correspond to the higher rate of keeping neurons, which each training step has more neurons to train. The red line is artificially removing neurons at too high a rate which slows it down, and so we can remove that option.

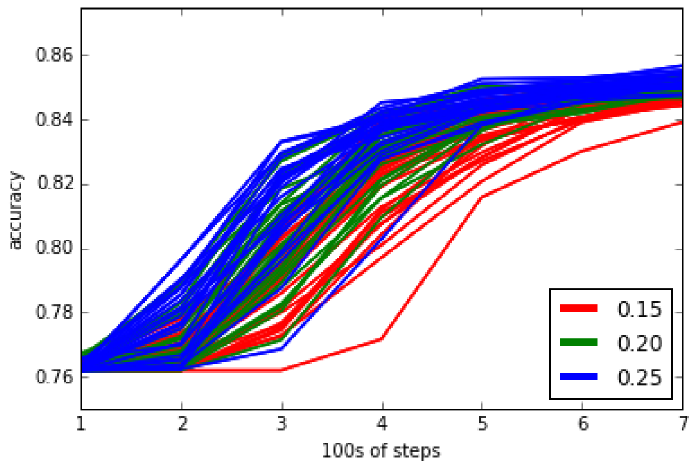

Illustration 13: Dropout rate

And the last parameter to look at is the width of the shallow part of the network. It is super hard to follow any of the lines since there is no banding. This probably corresponds to not having a strong correlation between this parameter and the speed at which it trains, quite possibly just random fluctuations coming from weight initialization.

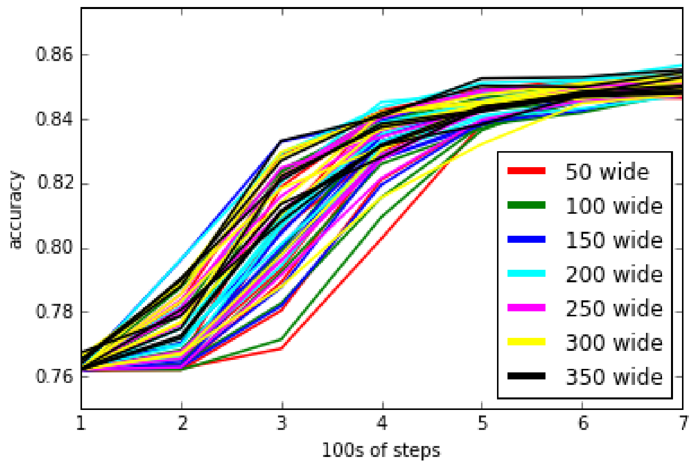

Illustration 14: Width of the shallow parts of the net

Conclusion

So after looking through the graphs, it seems like having only 3 layers on the deep part of the net is needed, but wider layers is better. Also having a dropout rate of 0.2 or 0.25 does not slow down the training rate too much. The decay rate of the width of the deep part and the width of the shallow part seem to not matter in any meaningful way. If we want to draw more conclusions, more data to test is needed, but due to time constraints that information can not be gathered for this paper.

Another conclusion is that using graphs to look at this data really made it easy to see what was going on. I have gotten kind of used to simply playing with parameters and going on intuition. But by piping hard data into graphs and then selecting what to colorize and show, it makes me a lot more confident in the conclusions that I draw. I see the use of the ipython notebook for this kind of exploratory work; you already have the data, but you want to screw around with what ways you look at it.

Also, numpy is hard to use as it fights with the normal list data type of python and you lose nearly all the normal tools for dealing with lists. I like simply having a flat memory space to work in and loops to not be super slow, but unfortunately tensor-flow only works properly with python. But numpy is a lot faster than the built-in lists and so is worth the effort of using them; it made the neural net training about 5x faster.

References

Websites
TensorFlow Wide and Deep Learning Tutorial: https://www.tensorflow.org/tutorials/wide_and_deep

Images:
Illustration 1 : http://cs231n.github.io/assets/nn1/neural_net.jpeg : A simple neural net example
Illustration 2 : https://www.tensorflow.org/images/wide_n_deep.svg : The deep and wide architecture

Code (should also be uploaded next to the report)

NeuralNet Code

```
import tensorflow as tf
from math import log
#from random import shuffle
from numpy import *
from numpy.random import choice,shuffle
import gc

class empty(): pass
nn = empty()

def make_onehot_dict(options_string):
      #returns a dictionary of prebuilt arrays
      #you pass in a comma seperated list of keys and we make buckets for them
      # "a,b,c" -> {"a":[1,0,0] , "b":[0,1,0] , "c":[0,0,1]}
      keys = options_string.split(",")
      temp = {"?":[0]*len(keys)}
      for k,i in zip(keys,range(len(keys))):
            k = k.strip()
            ar = [0]*len(keys)
            ar[i] = 1
            temp[k] = ar
      return temp
def scale(val,minn,maxx):
      #scales val to be from -1 to 1
      rangee = maxx - minn
      val -= minn
      val /= float(rangee) / 2.0
      return val - 1.0
def make_onehot_bucket_from_range(val,minn,maxx,num_buckets):
      #so if mapping 10 from 1 to 20 with 5 buckets
      #you get [0,0,1,0,0]
      rangee = maxx - minn
      if val<maxx: val -= minn
      else: val = rangee-1
      if val<0: val=0

      bucket_size = rangee/num_buckets
      ar = [0]*num_buckets
      ar[int(val//bucket_size)] = 1
      return ar

def parse_data_file(filename):
      f = open(filename,"r")
      age = lambda x: [scale(int(x),0,80)]
      age_buckets = lambda x: make_onehot_bucket_from_range(int(x),0,80,16)
      workclass = make_onehot_dict("Private, Self-emp-not-inc, Self-emp-inc,
Federal-gov, Local-gov, State-gov, Without-pay, Never-worked")
      fnlwgt = lambda x : [scale(int(x),0,500000)]
      fnlwgt_buckets = lambda x :
make_onehot_bucket_from_range(int(x),0,1000000,100)
      education = make_onehot_dict("Bachelors, Some-college, 11th, HS-grad,
Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th,
Doctorate, 5th-6th, Preschool")
      education_num = lambda x : [scale(int(x),0,20)]
      education_num_buckets = lambda x :
make_onehot_bucket_from_range(int(x),0,20,10)
```

```
      marital_status = make_onehot_dict("Married-civ-spouse, Divorced, Never-
married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse")
      occupation = make_onehot_dict("Tech-support, Craft-repair, Other-service,
Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct,
Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-
serv, Armed-Forces")
      relationship = make_onehot_dict("Wife, Own-child, Husband, Not-in-family,
Other-relative, Unmarried")
      race = make_onehot_dict("White, Asian-Pac-Islander, Amer-Indian-Eskimo,
Other, Black")
      sex = make_onehot_dict("Female, Male")
      capital_gain = lambda x: [scale(int(x),0,50000)]
      capital_gain_buckets = lambda x:
make_onehot_bucket_from_range(int(x),0,50000,50)
      capital_gain_buckets_log = lambda x:
make_onehot_bucket_from_range(int(math.log(int(x)+1,10)),0,9,10)
      capital_loss = lambda x: [scale(int(x),0,50000)]
      capital_loss_buckets = lambda x:
make_onehot_bucket_from_range(int(x),0,50000,50)
      capital_loss_buckets_log = lambda x:
make_onehot_bucket_from_range(int(math.log(int(x)+1,10)),0,9,10)
      hours_per_week = lambda x: [scale(int(x),0,80)] # assuming max number is
80 hours
      hours_per_week_buckets = lambda x:
make_onehot_bucket_from_range(int(x),0,120,12)
      native_country = make_onehot_dict("United-States, Cambodia, England,
Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece,
South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica,
Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador,
Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand,
Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands")

      income = make_onehot_dict("<=50K, >50K")

      data = []
      answers = []
      for line in f:
            line = line.split(",")
            line_data = array([],dtype=float32)
            try:
                  line_data = concatenate(( line_data , age(line[0].strip()) ))
                  line_data = concatenate(( line_data ,
age_buckets(line[0].strip()) ))
                  line_data = concatenate(( line_data ,
workclass[line[1].strip()] ))
                  line_data = concatenate(( line_data ,
fnlwgt(line[2].strip()) ))
                  line_data = concatenate(( line_data ,
fnlwgt_buckets(line[2].strip()) ))
                  line_data = concatenate(( line_data ,
education[line[3].strip()] ))
                  line_data = concatenate(( line_data ,
education_num(line[4].strip()) ))
                  line_data = concatenate(( line_data ,
education_num_buckets(line[4].strip()) ))
                  line_data = concatenate(( line_data ,
marital_status[line[5].strip()] ))
                  line_data = concatenate(( line_data ,
occupation[line[6].strip()] ))
                  line_data = concatenate(( line_data ,
relationship[line[7].strip()] ))
                  line_data = concatenate(( line_data , race[line[8].strip()] ))
                  line_data = concatenate(( line_data , sex[line[9].strip()] ))
                  line_data = concatenate(( line_data ,
capital_gain(line[10].strip()) ))
                  line_data = concatenate(( line_data ,
capital_gain_buckets(line[10].strip()) ))
                  line_data = concatenate(( line_data ,
capital_gain_buckets_log(line[10].strip()) ))
                  line_data = concatenate(( line_data ,
capital_loss(line[11].strip()) ))
                  line_data = concatenate(( line_data ,
capital_loss_buckets(line[11].strip()) ))
                  line_data = concatenate(( line_data ,
capital_loss_buckets_log(line[11].strip()) ))
                  line_data = concatenate(( line_data ,
hours_per_week(line[12].strip()) ))
                  line_data = concatenate(( line_data ,
hours_per_week_buckets(line[12].strip()) ))
                  line_data = concatenate(( line_data ,
native_country[line[13].strip()] ))
            except e:
                  print("Skipping line",e)
                  continue

            answers.append(array(income[line[14].strip()],dtype=float32))
            data.append(array(line_data,dtype=float32))
            # print(len(line_data))
            # print(line_data)
            # break

      data = array(data)
      answers = array(answers)
      return data,answers

#===============================================================================
================================================================================
================

def setup_layers(deep_sizes=[100,50,10],wide_size=100):
      full_layer = lambda i,x,y: tf.nn.relu(
                                                tf.matmul(i ,
tf.Variable(tf.truncated_normal([x,y],stddev=0.1)))

      +tf.Variable(tf.constant(0.1,shape=[y]))
                                          )

      input_size = nn.training_data.shape[1]
      output_size = nn.training_answers.shape[1]

      nn.input_= tf.placeholder(tf.float32,[None,input_size])
      nn.labels =tf.placeholder(tf.float32,[None,output_size])
      nn.dropout_rate_placeholder = tf.placeholder(tf.float32)

      deep = full_layer( nn.input_, input_size,deep_sizes[0] )
      for index in range(1,len(deep_sizes)):
            deep = full_layer( deep, deep_sizes[index-1],deep_sizes[index] )
      wide = full_layer( nn.input_, input_size,wide_size )
      total= tf.concat([deep,wide],axis=1)
      total= tf.nn.dropout(total,nn.dropout_rate_placeholder)

      nn.output_ = tf.matmul(total ,
tf.Variable(tf.truncated_normal([deep_sizes[-
1]+wide_size,output_size],stddev=0.1)))

      loss = tf.losses.sigmoid_cross_entropy(nn.labels,nn.output_)
      nn.optimizer = tf.train.AdamOptimizer(0.0001).minimize(loss)
      nn.accuracy = tf.reduce_mean(tf.cast( tf.equal(tf.argmax(nn.output_, 1),
tf.argmax(nn.labels, 1)) , tf.float32))


#================================================================================
================================================================================
================

def train_nn():
      sess = tf.Session()
      sess.run(tf.global_variables_initializer())
      training_loss=[]
      for epoch in range(nn.epochs+1):
            indexes = choice(nn.training_data.shape[0],250)
            batch_data = nn.training_data[indexes]
            batch_answers = nn.training_answers[indexes]

            sess.run(nn.optimizer,feed_dict =
{nn.input_:batch_data,nn.labels:batch_answers,nn.dropout_rate_placeholder:nn.dro
pout_rate})
            if epoch%50 == 0:
            #     print(epoch,sess.run(nn.accuracy,feed_dict =
{nn.input_:nn.validation_data[0],nn.labels:nn.validation_data[1],nn.dropout_rate
_placeholder:1}))
                  training_loss.append(sess.run(nn.accuracy,feed_dict =
{nn.input_:nn.validation_data,nn.labels:nn.validation_answers,nn.dropout_rate_pl
aceholder:1}))


      #and now finally the test data
      #acc = sess.run(nn.accuracy,feed_dict =
{nn.input_:nn.test_data[0],nn.labels:nn.test_data[1],nn.dropout_rate_placeholder
:1})
      #print("\nfinal accuracy",acc)
      tf.reset_default_graph()
      sess.close()
      return training_loss

#================================================================================
================================================================================
================

def run_test():
      nn.training_data,nn.training_answers = parse_data_file("adult.data")
      nn.training_data,nn.training_answers =
ill_just_shuffle_it_myself(nn.training_data,nn.training_answers)

      validation_size       = int( nn.training_data.shape[0] * 0.15 )
      nn.validation_data    = nn.training_data[:validation_size]
      nn.validation_answers = nn.training_answers[:validation_size]
      nn.training_data      = nn.training_data[validation_size:]
      nn.training_answers   = nn.training_answers[validation_size:]
      del validation_size

      #nn.test_data = parse_data_file("adult.test")
      #nn.test_data = array(zip(*nn.test_data))
      nn.epochs = 2000

      f = open("results.csv","w")

f.write("Depth,Depth_Decay,Depth_Start,Dropout_Rate,Width,"+numList_string(nn.ep
ochs,50)+"\n")
      f.flush()
      count=0
      for depth in range(3,7):
            for depth_decay in [x/10.0 for x in range(4,6)]: #range(0.1,0.8,.1)
                  for depth_start in range(50,400,50):
                        deep_sizes = [int(depth_start*(depth_decay**x)) for x in
range(depth)]
                        for dropout_rate in [x/100.0 for x in range(15,30,5)]:
#range(0.05,0.5,0.05)
                              nn.dropout_rate = dropout_rate
                              for width in range(50,400,50):
                                    setup_layers(deep_sizes,width)
                                    training_loss = train_nn()
                                    count+=1
                                    print("did round",count)


f.write(make_csv_line(depth,depth_decay,depth_start,dropout_rate,width,training_
loss)+"\n")
                                    f.flush()
                                    gc.collect()
      print("done")

def numList_string(size,step):
      s=""
      for x in range(0,size+1,step):
            s+="epoch"
            s+=str(x)
            s+=","
      return s[:-1] #remove the last comma

def
make_csv_line(depth,depth_decay,depth_start,dropout_rate,width,training_loss):
      s=""
      s+=str(depth)+","
      s+=str(depth_decay)+","
      s+=str(depth_start)+","
      s+=str(dropout_rate)+","
      s+=str(width)+","
      for x in training_loss:
            s+=str(x)+","
      return s[:-1] #remove that last comma

def ill_just_shuffle_it_myself(a1,a2):
      length = a1.shape[0]
      ind=arange(length)
      shuffle(ind)
      a1 = a1[ind]
      a2 = a2[ind]
      return a1,a2


if(__name__ == "__main__"):
      run_test()
```

NoteBook Code

```
# coding: utf-8

# In[15]:

get_ipython().magic('pylab inline')
```

```
from numpy import *
import pandas as pd
import matplotlib.pyplot as plt


# In[151]:

f = open("results.csv","r")
data = pd.read_csv(f)
f.close
del f


# Before we start looking at individual parameteres, lets see how much variance
there is in the data. If there is nearly no change in the way it behaves with
all of the changes, then it would be best to simply take the simplist set and
move on. But as we can see in the graph below, there is difference between the
maximum values and the minimum values forr all the data. This probably means
that some parameters are making a useful difference in how it behaves.

# In[175]:

def max_from_each_epoch(dd):
    maxx = []
    for x in dd:
        if "epoch" in x:
            maxx.append(max(dd[x]))
    return maxx
def min_from_each_epoch(dd):
    minn = []
    for x in dd:
        if "epoch" in x:
            minn.append(min(dd[x]))
    return minn
def get_training_data_only(dd):
    ar = []
    for x in dd:
        if "epoch" in x:
            ar.append(dd[x])
    ar = array(ar)
    return ar.T
def artist_plot_legend_first(ax,ar,label,color):
    ax.plot(ar[0],label=label,color=color)
    ax.plot(ar[1:].T,label="_nolegend_",color=color)


# In[105]:

plt.plot(max_from_each_epoch(data))
plt.plot(min_from_each_epoch(data))
plt.ylabel("accuracy")


# To start looking at parameters, lets start by comparing the max training of
each layer individually. As you can see, there is no real difference in the
final accuracy, but the shallower layers get to the maximum sooner. 6 layers is
notably slower than the others to get the last couple percentage, not getting to
the max until about 3000 training steps, while 3 and 4 deep layers are about the
same, reaching the maximum aorund 2500 training steps

# In[112]:

fig,ax = plt.subplots()
ax.plot(max_from_each_epoch(data[data.Depth == 3]),label="depth 3")
ax.plot(max_from_each_epoch(data[data.Depth == 4]),label="depth 4")
ax.plot(max_from_each_epoch(data[data.Depth == 5]),label="depth 5")
ax.plot(max_from_each_epoch(data[data.Depth == 6]),label="depth 6")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)


# Well, since 3 layers seems deep enough, lets see how much variance it has
itself. Interesting thing to note is the oddly bulgy variance at the beginning,
but it seems to converge after roughly 1500 training steps in. Also note it gets
very small variance between min and max by 2500 steps; I would say that there is
no point in training further than that with only 3 layers no matter the other
parameters we choose.

# In[114]:

data_layer3 = data[data.Depth == 3]
fig,ax = plt.subplots()
ax.plot(max_from_each_epoch(data_layer3),label="max")
ax.plot(min_from_each_epoch(data_layer3),label="min")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)


# Lets now look at all the different options at once and see if there is any
major divide in the data. If there is, then that means there is likely a single
factor that is simply worse in one state. Looking at the graph, besides being a
giant rainbow, the lines seem to simply converge to the same maximum with a
fairly even spread, even in the buldge on the left. That means there does not
seem to be any option that is binary good or bad.

# In[110]:

data_layer3 = data[data.Depth == 3]
fig,ax = plt.subplots()
ax.plot(get_training_data_only(data_layer3).T)
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
for l in ax.lines:
    l.set_linewidth(1.5)


# Since there is not much difference in the maximum accuracy (likely only
varying due to random weight initialisations) then the other factor left to
optimize is training time. Lets start by taking look at the most obvious thing
to affect the time, how wide the starting layer for the deep side of the net is.
Interesting to note, is that the largest starting size is the fastest to
converge, not really ever having the bulge area affect it.
#
# Sorry for not having a legend, but i got tired of playing with matplotlib on
that front.

# In[194]:
```

```
data_layer3 = data[data.Depth == 3]
starting_sizes = []
for x in range(50,400,50):

starting_sizes.append(get_training_data_only(data_layer3[data_layer3.Depth_Start
== x]))

fig,ax = plt.subplots()
artist_plot_legend_first(ax,starting_sizes[0],"50 wide","red")
artist_plot_legend_first(ax,starting_sizes[1],"100 wide","green")
artist_plot_legend_first(ax,starting_sizes[2],"150 wide","blue")
artist_plot_legend_first(ax,starting_sizes[3],"200 wide","cyan")
artist_plot_legend_first(ax,starting_sizes[4],"250 wide","magenta")
artist_plot_legend_first(ax,starting_sizes[5],"300 wide","yellow")
artist_plot_legend_first(ax,starting_sizes[6],"350 wide","black")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)


# Though we think we know exactly what is going on, lets trim it down to zoom in
on the bulge area to see more details. This graph simply has the x and y axes
zoomed in so that we can try to see the groups more clearly. The top left graph
shows all the groups, which looks a little messy still, but we can see that the
red lines are probably the worst off. The red group is the smallest number of
starting neurons in the deep part of the network. In the graph to the right of
it, it is easy to see that no lines of the red are close to the black, so we can
just outright remove that option.
#
# The next two graphs show different groups compared to the black. Blue and
green show an improvment over the red, but are in the same situation where they
are not close to black, and so we can remove those options also. The yellow and
pink lines are touching or covered up by the black lines, but pink is obviously
sticking out a bit so we can remove that option, leaving yellow. Yellow seems to
be mostly covered up by the black lines, and so it might be best to leave the
option until we look at the other parameters. This will help us tease out what
other parameters make a difference in training time by comparing how well the
options work between these two groups.
#
# *take the next 4 graphs and make a square with them

# In[193]:

data_layer3 = data[data.Depth == 3]
starting_sizes = []
for x in range(50,400,50):

starting_sizes.append(get_training_data_only(data_layer3[data_layer3.Depth_Start
== x]))

fig,ax = plt.subplots()
artist_plot_legend_first(ax,starting_sizes[0],"50 wide","red")
artist_plot_legend_first(ax,starting_sizes[1],"100 wide","green")
artist_plot_legend_first(ax,starting_sizes[2],"150 wide","blue")
artist_plot_legend_first(ax,starting_sizes[3],"200 wide","cyan")
artist_plot_legend_first(ax,starting_sizes[4],"250 wide","magenta")
artist_plot_legend_first(ax,starting_sizes[5],"300 wide","yellow")
artist_plot_legend_first(ax,starting_sizes[6],"350 wide","black")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
```

```
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,25])


# In[192]:

data_layer3 = data[data.Depth == 3]
starting_sizes = []
for x in range(50,400,50):

starting_sizes.append(get_training_data_only(data_layer3[data_layer3.Depth_Start
== x]))

fig,ax = plt.subplots()
artist_plot_legend_first(ax,starting_sizes[0],"50 wide","red")
artist_plot_legend_first(ax,starting_sizes[6],"350 wide","black")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,25])


# In[191]:

data_layer3 = data[data.Depth == 3]
starting_sizes = []
for x in range(50,400,50):

starting_sizes.append(get_training_data_only(data_layer3[data_layer3.Depth_Start
== x]))

fig,ax = plt.subplots()
artist_plot_legend_first(ax,starting_sizes[1],"100 wide","green")
artist_plot_legend_first(ax,starting_sizes[2],"150 wide","blue")
artist_plot_legend_first(ax,starting_sizes[6],"350 wide","black")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,25])


# In[190]:

data_layer3 = data[data.Depth == 3]
starting_sizes = []
for x in range(50,400,50):

starting_sizes.append(get_training_data_only(data_layer3[data_layer3.Depth_Start
== x]))
```

```
fig,ax = plt.subplots()
artist_plot_legend_first(ax,starting_sizes[4],"250 wide","magenta")
artist_plot_legend_first(ax,starting_sizes[5],"300 wide","yellow")
artist_plot_legend_first(ax,starting_sizes[6],"350 wide","black")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,25])


# At this point, we know that we are near the top in how quickly we can train
the net. For the most part, the next parameters do not seem to make a big
difference in how quickly it trains and really we are nitpicking. But lets take
a look anyways, and see if there are any conclusions we can draw.
#
# This first graph is what i am calling the decay rate for the width of the
layers in the deep part of the net. Due to time constraints, only two values
were tested, but we can see that the green lines seem to be slightly faster to
train on average, which corisponds to wider layers in general on the deep side.
The red lines are not really much worse, and so really probably doesn't matter
much.

# In[184]:

data_layer3 = data[data.Depth == 3]
data_layer3 = data_layer3[(data_layer3.Depth_Start == 300) |
(data_layer3.Depth_Start == 350)]
depth_decay = []
for x in [x/10.0 for x in range(4,6)]:

depth_decay.append(get_training_data_only(data_layer3[data_layer3.Depth_Decay ==
x]))

fig,ax = plt.subplots()
ax.plot(depth_decay[0].T,color="red")
ax.plot(depth_decay[1].T,color="green")
artist_plot_legend_first(ax,depth_decay[0],"0.4","red")
artist_plot_legend_first(ax,depth_decay[1],"0.5","green")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,7])


# This next graph shows the 3 different tested dropout rates. We can see that
the blue and green lines are mostly in the same area, but the red lines are
noticably slower. The faster rates corispond to the higher rate of keeping
neurons, which each training step has more neurons to train. The red line is
artifically removing nerons at too high a rate which slows it down, and so we
can remove that option.

# In[185]:
```

```
data_layer3 = data[data.Depth == 3]
data_layer3 = data_layer3[(data_layer3.Depth_Start == 300) |
(data_layer3.Depth_Start == 350)]
dropout_rate = []
for x in [x/100.0 for x in range(15,30,5)]:

dropout_rate.append(get_training_data_only(data_layer3[data_layer3.Dropout_Rate
== x]))

fig,ax = plt.subplots()
ax.plot(dropout_rate[0].T,color="red")
ax.plot(dropout_rate[1].T,color="green")
ax.plot(dropout_rate[2].T,color="blue")
artist_plot_legend_first(ax,dropout_rate[0],"0.15","red")
artist_plot_legend_first(ax,dropout_rate[1],"0.20","green")
artist_plot_legend_first(ax,dropout_rate[2],"0.25","blue")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,7])


# And the last parameter to look at is the width of the shallow part of the
network. It is super hard to follow any of the lines since there is no banding.
This probably coresponds to not having a strong corelation between this
parameter and the speed at which it trains, quite possibly just random
fluctuations coming from weight initialisations.

# In[188]:

data_layer3 = data[data.Depth == 3]
data_layer3 = data_layer3[(data_layer3.Depth_Start == 300) |
(data_layer3.Depth_Start == 350)]
data_layer3 = data_layer3[(data_layer3.Dropout_Rate == 0.20) |
(data_layer3.Dropout_Rate == 0.25)]
width_options = []
for x in range(50,400,50):
    width_options.append(get_training_data_only(data_layer3[data_layer3.Width ==
x]))

fig,ax = plt.subplots()
artist_plot_legend_first(ax,width_options[0],"50 wide","red")
artist_plot_legend_first(ax,width_options[1],"100 wide","green")
artist_plot_legend_first(ax,width_options[2],"150 wide","blue")
artist_plot_legend_first(ax,width_options[3],"200 wide","cyan")
artist_plot_legend_first(ax,width_options[4],"250 wide","magenta")
artist_plot_legend_first(ax,width_options[5],"300 wide","yellow")
artist_plot_legend_first(ax,width_options[6],"350 wide","black")
plt.ylabel("accuracy")
plt.xlabel("100s of steps")
leg = ax.legend(loc="lower right")
for l in ax.lines:
    l.set_linewidth(1.5)
for l in leg.get_lines():
    l.set_linewidth(4)
plt.ylim([0.75,0.875])
plt.xlim([1,7])
```

```
# Conclusion
# we found some params. Graphs do help when looking at this stuff. I am not an
expert on neural nets and so i could be wildy wrong about most of this stuff.
```