Dec 12th 2017

Zirui Ding

Harvard Extensio School Big Data Project

Neural Network Application in Finance

Neural Network Default Model

# Abstract

The project focus on building neural network to predict the default probability. Also, it provides the opportunity to explore how different neural structures can impact the accuracy and efficiency of the model.

With 1-hidden layer model, the neural network has already out performed most of the logistic regression model with accuracy of 92.3%. I adjust the number of neurons in the hidden layer. I notice that the model training efficiency does not follow a monotonic relationship with the number of neurons. The training time initially increases with the number of neurons, then decreases and then increases again.

I further build model using 2-hidden layers and 3-hidden layers model. However, the model accuracy does not have significant improvement. When plotting out the testing set accuracy and loss, it seems that the model may suffer from over fit. I applied dropout function to handle the overfat.

It is an explorative exercise to use neural network to build mortgage default prediction model. It seems that the neural network does have strong prediction power. However, the improvement of such model may not be easy. 1,2 and 3 hidden layer models produced similar results for the fully connected model. Maybe other structures can be explored to boost the efficiency and accuracy of the model.

# Background

Traditional default models used by financial institutions are built based on the logistic regression structure. Due to the limited flexibility in such structure, the model can only capture the limited information from the existing data. It’s prediction power is limited. Usually, any financial institution would apply multiple logistic default model with various input and structure in order to better estimate the entity default rate. Neural network, on the other hand, provides a flexible structure to take on large sample of data and modeling complex relationship between input variables and the target prediction. In this case study, I try to build neural network on mortgage data to predict the commercial mortgage default rate.

# Data and Pre-processing

The original data were sourced from Kaggle, provided by Lendging Club. It is publicly available on the site <https://www.kaggle.com/wendykan/lending-club-loan-data>. The dataset itself contains loan payment information concerning individual loans issues by lending club from 2007 to 2015. Dataset include information from basic loan characters to borrower’s credit records and occupation. There are totally 47 features available including the loan status (our label variable).

Concerning the limited time and constrained computational power, I started the project with 8 input variables, namely home ownership, number of queries in last 6-month, purpose of the loan, term of the loan, interest rate, annual income and earliest credit line. Within these variables, number of queries in last 6-month, term of the loan, interest rate and annual income are numeric variables that we can directly applied to the model.

Home ownership and purpose of the loan are the two categorical data here. I created the hashtable to map each category onto the numeric number and convert the two variables accordingly. Earliest credit line is the datetime format. I picked the earliest date in this variable as the baseline date. All other dates are then calculated as the number of days lapsed from the baseline date. In this way, I can convert the date time format into the continuous numeric variable format.

For default, I refer to the column of “loan\_status”. All value shown as “Fully Paid”, “current”, “Issued” or “Does not meet the credit policy. Status:Fully Paid” are marked as non default loans. The rest are loans defaulted. The column will be used as the label columns later in our model training.

After call these conversion, the transformed data set is saved as “Cleaned data.csv” in the system. It helps to save the limited memory space on the laptop.

The detailed code is not shown here since it is not the core code for this project. But it can be referred to in the iPython notebook I submitted named “Data Preprocessing”.

Every time before the data is applied, it will be randomly split into three parts. 70% of the data are used as training set, 15% of the data is used as testing set and the remaining 15% is put aside as validation set. The code is shown below:

import numpy as np

np.random.seed(10)

random\_number=np.random.rand(len(dataset.index.values))

msk1=random\_number<0.7

msk2=random\_number>0.85

msk3=(random\_number>=0.7) & (random\_number<=0.85)

train\_set=dataset.loc[msk1,:]

test\_set=dataset.loc[msk2,:]

validation\_set=dataset.loc[msk3,:]

Also, I considered scaling the data to pull them to the same level. Some of the data, such as annual income is particularly big when comparing with data such as interest rate. In order to scale the data, sklearn package is used. 4 variables, namely loan amount, interest rate, annual income and earliest credit line are being rescaled to zero mean and unit variance data. The code is shown as below.

#design the function to resclae the data to adjust the loan amount, interest rate, annual income and earliest credit line

from sklearn import preprocessing

def scale\_train\_data(df):

target=['loan\_amnt','int\_rate','annual\_inc','earliest\_cre\_line']

new\_df=df.loc[:,df.columns.difference(target)]

loan\_amount\_scaler=preprocessing.StandardScaler().fit(df[target[0]].reshape(-1, 1))

new\_df['scaled\_'+target[0]]=loan\_amount\_scaler.transform(df[target[0]].reshape(-1, 1))

int\_rate\_scaler=preprocessing.StandardScaler().fit(df[target[1]].reshape(-1, 1))

new\_df['scaled\_'+target[1]]=int\_rate\_scaler.transform(df[target[1]].reshape(-1, 1))

annual\_inc\_scaler=preprocessing.StandardScaler().fit(df[target[2]].reshape(-1, 1))

new\_df['scaled\_'+target[2]]=annual\_inc\_scaler.transform(df[target[2]].reshape(-1, 1))

earliest\_cre\_line\_scaler=preprocessing.StandardScaler().fit(df[target[3]].reshape(-1, 1))

new\_df['scaled\_'+target[3]]=earliest\_cre\_line\_scaler.transform(df[target[3]].reshape(-1, 1))

return new\_df,[loan\_amount\_scaler,int\_rate\_scaler,annual\_inc\_scaler,earliest\_cre\_line\_scaler]

#design the function to apply the scaler onto test and validation set

def apply\_scaler(df,scaler\_list):

target=['loan\_amnt','int\_rate','annual\_inc','earliest\_cre\_line']

new\_df=df.loc[:,df.columns.difference(target)]

for i in range(len(target)):

new\_df['scaled\_'+target[i]]=scaler\_list[i].transform(df[target[i]].reshape(-1, 1))

return new\_df

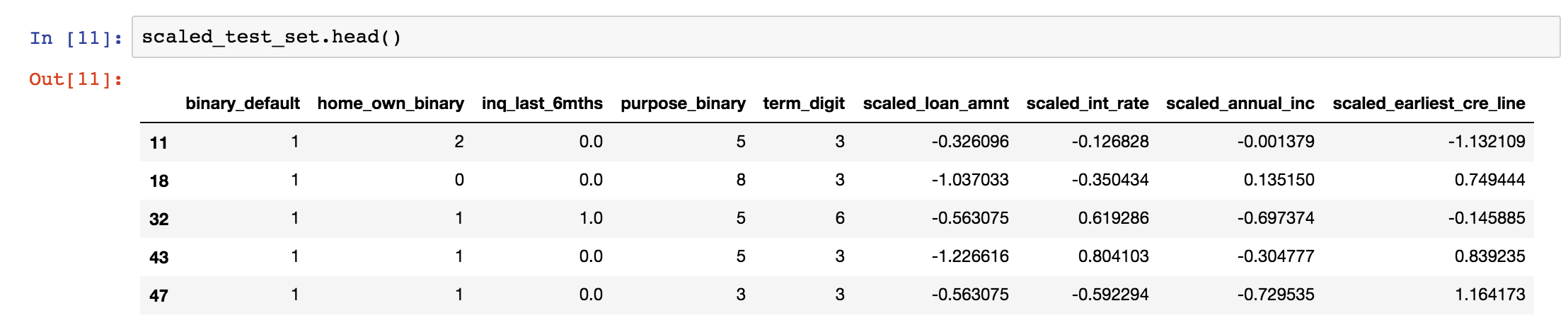
scaled\_train\_set,scaler\_list=scale\_train\_data(train\_set)

scaled\_test\_set=apply\_scaler(test\_set,scaler\_list)

scaled\_validation\_set=apply\_scaler(validation\_set,scaler\_list)

The first function scales the training data and return the four scaler. The second function use the four scalers to convert the data in the testing and validation set to make sure the scaling is the same across different data sets.

An example of the scaled data is provided here:



# Demo and Working Code

## Single Hidden Layer Model with not scaling

I start from the most basic structure by building a fully connected neural network with 1-hidden layer. A diagram of the model layer is shown below.

…

…

\_

\_

\_

Input Layer

(Loan Characters)

Hidden Layer

(10 nodes)

Output Layer

Softmax

In this simplest model, the eight input variables are first mapped to a hidden layer of 10 neurons. By linear transformation. At each neuron, the activation function relu is used to transform the signal from bottom layer to upper layer.

At the output layer, the linear combination from signals from hidden layer is calculated and pass through softmax function to calculate the probability for the two cases (default or not default).

The TensorFlow code is shown below:

import tensorflow as tf

#set placeholder for

input\_x=tf.placeholder(dtype=tf.float32,shape=(None,8))

input\_y=tf.placeholder(dtype=tf.int32,shape=(None))

#set variables to calculate the number of

n\_neural=10

#initialize the transfer matrix and bias term

w1 = tf.Variable(tf.truncated\_normal([8, n\_neural], stddev=0.1), name="W")

b1=tf.Variable(tf.zeros([n\_neural]), name="B",dtype=tf.float32)

w2 = tf.Variable(tf.truncated\_normal([n\_neural,2], stddev=0.1), name="W")

b2=tf.Variable(tf.zeros([2]), name="B",dtype=tf.float32)

#construct the hidden layer layer

net=tf.nn.relu(tf.add(tf.matmul(input\_x,w1),b1))

#output layer

logits=tf.add(tf.matmul(net,w2),b2)

#construct the loss function

loss=tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=logits,labels=input\_y))

#optimization fucntion

learning\_rate = 0.001

optimizer = tf.train.AdamOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

probability=tf.nn.softmax(logits)

classification=tf.nn.in\_top\_k(logits, input\_y, 1)

accuracy=tf.reduce\_mean(tf.cast(classification,tf.float32))

There are a few more points I need to elaborate in the parameters setting and function/optimizer selection process.

* AdamOptimizer is used instead of the GradientDescentOptimizer. I initially used GradientDescentOptimizer follwoing most of the examples in our classes. However, I notice that this optimizer provide almost the fixed result when changing neuron numbers, layer numbers or even the keep probably in the dropout function. Result from this optimizer is rather insensitive to the change in model structure. I searched online that such case may happen due to the data characters. A recommended approach is to change the optimizer. After reading Aurélien Géron’s book Chapter 11, I adjust the optimizer to AdamOptimizer with a more robust approach to optimize the neural network.
* Learning rate is set to 0.001, which is the default learning rate for AdamOptimizer.
* Softmax function is not directly applied at the output layer. So we can use the function sparse\_softmax\_cross\_entropy\_with\_logits(). The function is equivalent to applying the softmax function and then computing the cross entropy. But it is more efficient to handle corner cases.
* The training data set is split into batches of 50 and send to train the model.

The below code is used to run the model and test the model on the testing and validation set. Note that the input data here is directly read from cleaned data file with no scaling applied.

init = tf.global\_variables\_initializer()

batch\_size=50

with tf.Session() as sess:

    init.run()

    for iteration in range(len(train\_set.index.values)//batch\_size):

        X=train\_set.iloc[(iteration\*batch\_size):((iteration+1)\*batch\_size),0:8]

        y=train\_set.iloc[(iteration\*batch\_size):((iteration+1)\*batch\_size),8]

        lo=sess.run([training\_op],feed\_dict={input\_x:X,input\_y:y})

    #running test set and validation set

    test\_prediction,test\_accuracy=sess.run([probability,accuracy],feed\_dict={input\_x:test\_set.iloc[:,0:8],input\_y:test\_set.iloc[:,8]})

    print(list(zip(test\_set.iloc[:,8],test\_prediction))[0:10])

print(test\_accuracy)

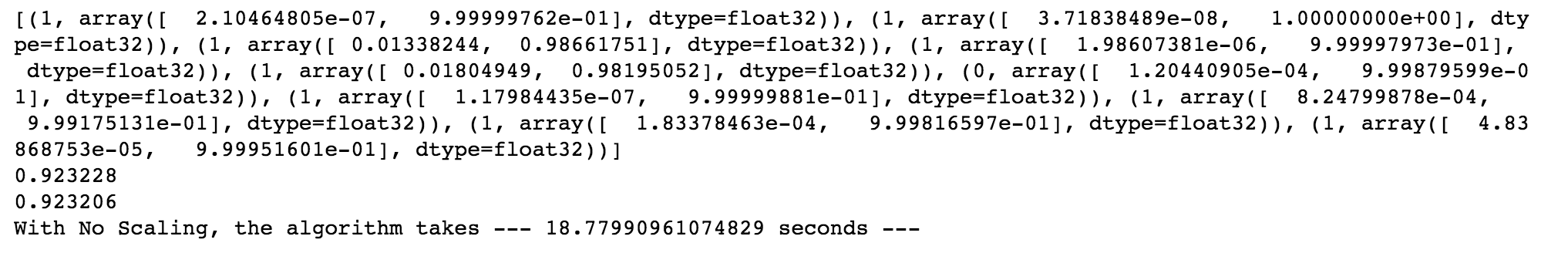
    validate\_prediction,validate\_accuracy=sess.run([probability,accuracy],feed\_dict={input\_x:validation\_set.iloc[:,0:8],input\_y:validation\_set.iloc[:,8]})

    #print(zip(validation\_set.iloc[:,8],validate\_prediction))

    print(validate\_accuracy)

sess.close()

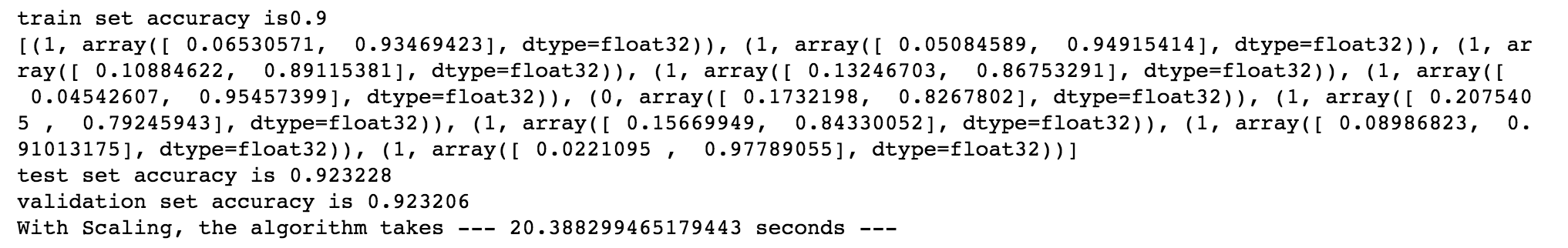
The code prints out the testing set accuracy together with 10 example of output. It also print the validation set accuracy at the last time. The result is screen captured here:



The model with a single hidden layer already has strong prediction power. The prediction accuracy for the model on the testing set and validation set are both around 92.3%, which is pretty impressive comparing with most of the logistic regression models used by the financial institutes to measure default.

## Single Hidden Layer Model with scaling

I then used the scaled dataset to rerun the function. I try to compare if scaling has any effect to improvement the model performance. The model output is shown here:



After scaling the variables, the 10-node 1 hidden layer model provides the same accuracy. However, the running time is 2 seconds behind when comparing with the initial model. It seems that the scaling may not necessarily increase the model efficiency in this case. However, the technique will be very beneficial when different data are used at different units. It is always a robust way to run the data scaling before applying to the model. In the rest of the discussion, all models are run using the scaled version of the dataset which went through exactly the same scaling procedure introduced previously.

## Single Hidden Layer Model with Varying Number of Neurons

I am curious to know how the number of neurons impact the model performance and efficiency. So I went one step further to run several different models by changing the number of neurons. Below code is designed to provide changing number of neurons to the model.

for n in [4,6,8,10,12,14,16]:

    hidden=construct\_neural\_layer(n,input\_x,"Hidden\_Layer",activation\_fnc="relu")

    output=construct\_neural\_layer(2,hidden,"Output\_Layer")

    loss=tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=input\_y,logits=output))

    accuracy=tf.reduce\_mean(tf.cast(tf.nn.in\_top\_k(output,input\_y,1),tf.float32))

    with tf.name\_scope("train"):

        optimizer = tf.train.AdamOptimizer(learning\_rate)

        training\_op = optimizer.minimize(loss)

    init = tf.global\_variables\_initializer()

    start\_time=time.time()

    with tf.Session() as sess:

        init.run()

        for iters in range(len(scaled\_train\_set.index.values)//batch\_size):

            x=scaled\_train\_set.iloc[(iters\*batch\_size):((iters+1)\*batch\_size),1:9]

            y=scaled\_train\_set.iloc[(iters\*batch\_size):((iters+1)\*batch\_size),0]

            train\_loss,train\_accuracy=sess.run([training\_op,accuracy],feed\_dict={input\_x:x,input\_y:y})

            #add summary if needed later

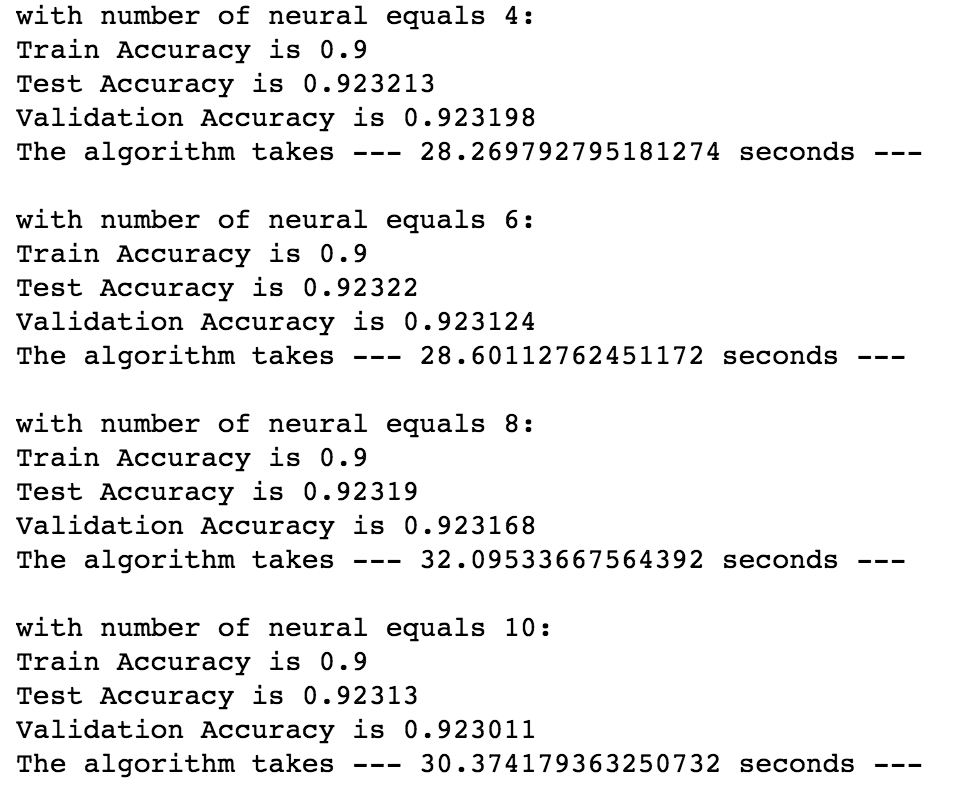
        acc\_train=accuracy.eval(feed\_dict={input\_x:x,input\_y:y})

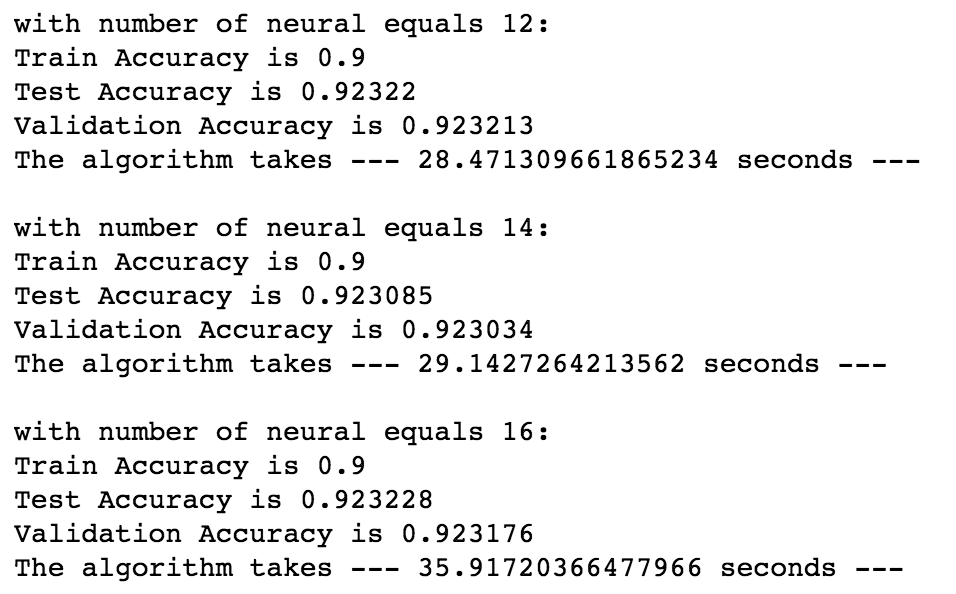
        acc\_test=accuracy.eval(feed\_dict={input\_x:scaled\_test\_set.iloc[:,1:9],input\_y:scaled\_test\_set.iloc[:,0]})

        acc\_valid=accuracy.eval(feed\_dict={input\_x:scaled\_validation\_set.iloc[:,1:9],input\_y:scaled\_validation\_set.iloc[:,0]})

        sess.close()

The model uses different number of neurons from 4 to 16 to rerun the model with scaled dataset. My initial guess was that as the neurons increase, the time to train such model should increase. Also, the prediction accuracy should also go up. However, the actual results surprise me a lot.





Indeed, 16 neuron model takes most of the time to train. However, 14-neuron model trains faster than the 8-neuron model. The relationship between time and number of neurons is not monotonic.

The accuracy of the model remains roughly unchanged with varying number of neurons.

## Two Hidden Layers Model without Dropout

I go on further two explore if increasing the number of layer can help improve the model performance. The structure of the two hidden layers model is quite similar to the one layer one. The below picture shows its structure.



The input layer and the first hidden layer is exactly the same as the single hidden layer model. In the second hidden layer, there are only 5 neurons available. I try to create this funnel structure to extract the key features gradually. Relu function is again applied in the second layer as the activate function. Output from second hidden layer is sent to the output layer together with softmax function to calculate the probability of default or not.

In order to ease the coding process, I designed a function to help me create layers.

import tensorflow as tf

import numpy as np

def construct\_neural\_layer(input\_x, layer\_nn,name, activation\_fnc=None):

    with tf.name\_scope(name):

        n\_var=int(input\_x.get\_shape()[1])

        #create random weights

        stddev=2/np.sqrt(n\_var)

        w=tf.Variable(tf.truncated\_normal((n\_var,layer\_nn),stddev=stddev),name="weight",dtype=tf.float32)

        b=tf.Variable(tf.zeros([layer\_nn]),name="biases",dtype=tf.float32)

        output=tf.add(tf.matmul(input\_x,w),b)

        if activation\_fnc=="sigmoid":

            return tf.sigmoid(output)

        elif activation\_fnc=="relu":

            return tf.nn.relu(output)

        elif activation\_fnc=="softmax":

            return tf.nn.softmax(output)

        else:

            return output

The basic structure is still the same with one hidden layer model except that the second hidden layer of 5 neurons are added. The function requires input variables, the output neuron numbers, name of the layer as well as the activation function. There are a few points I want to mention:

* Standard derivation of the random number is set to be 2/sqrt(num\_of\_variable). I checked the reference, it seems that setting standard derivation to be can make the algorithm more efficient according to Professor Aurélien Géron in his book Hands-on Machine Learning with Scikit-Learn and TensorFlow (ed.1, chapter 11).
* The activation\_fnc input is designed to bring flexibility to pick the activation function. By doing so, we can also use the function to construct the output layer because this layer does not require to use relu function.

The code used to create the layers and train the data is shown below:

#set the variables number for two layers

layer1\_nn=10

layer2\_nn=5

input\_x=tf.placeholder(tf.float32,shape=(None,8),name="X")

input\_y=tf.placeholder(tf.int32,shape=(None),name="y")

batch\_size=50

learning\_rate=0.01

hidden1=construct\_neural\_layer(input\_x,layer1\_nn,"Hidden\_Layer1",activation\_fnc="relu")

hidden2=construct\_neural\_layer(hidden1,layer2\_nn,"Hidden\_Layer2",activation\_fnc="relu")

output=construct\_neural\_layer(hidden2,2,"Output\_Layer")

loss=tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=input\_y,logits=output))

accuracy=tf.reduce\_mean(tf.cast(tf.nn.in\_top\_k(output,input\_y,1),tf.float32))

tf.summary.scalar('loss', loss)

tf.summary.scalar('accuracy', accuracy)

merged = tf.summary.merge\_all()

train\_writer = tf.summary.FileWriter('NN2\_no\_dropout')

with tf.name\_scope("train"):

    optimizer = tf.train.AdamOptimizer(learning\_rate)

    training\_op = optimizer.minimize(loss)

init = tf.global\_variables\_initializer()

with tf.Session() as sess:

    init.run()

    for iters in range(len(scaled\_train\_set.index.values)//batch\_size):

        X=scaled\_train\_set.iloc[(iters\*batch\_size):((iters+1)\*batch\_size),1:9]

        Y=scaled\_train\_set.iloc[(iters\*batch\_size):((iters+1)\*batch\_size),0]

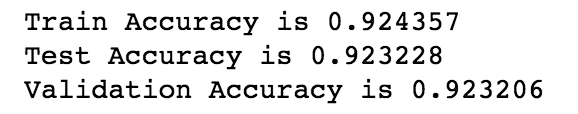
        opt,train\_loss,train\_accuracy,summ=sess.run([training\_op,loss,accuracy,merged],feed\_dict={input\_x:X,input\_y:Y})

        train\_writer.add\_summary(summ,iters\*batch\_size)

        test\_loss,test\_accuracy=sess.run([loss,accuracy],feed\_dict={input\_x:scaled\_test\_set.iloc[:,1:9],input\_y:scaled\_test\_set.iloc[:,0]})

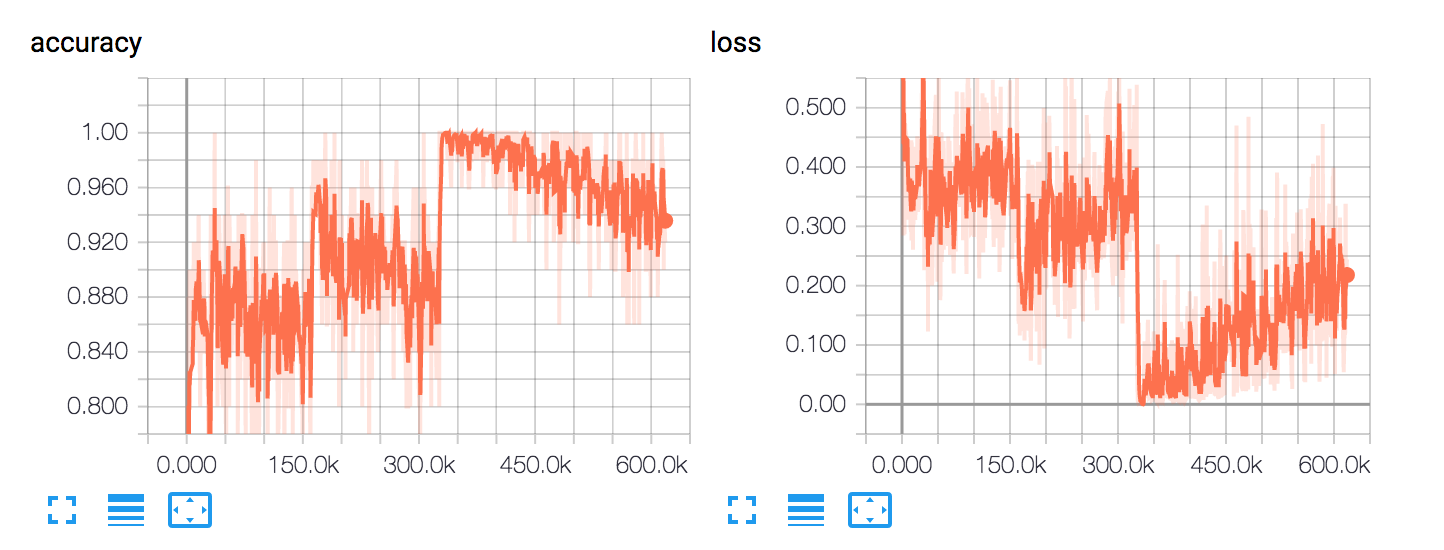
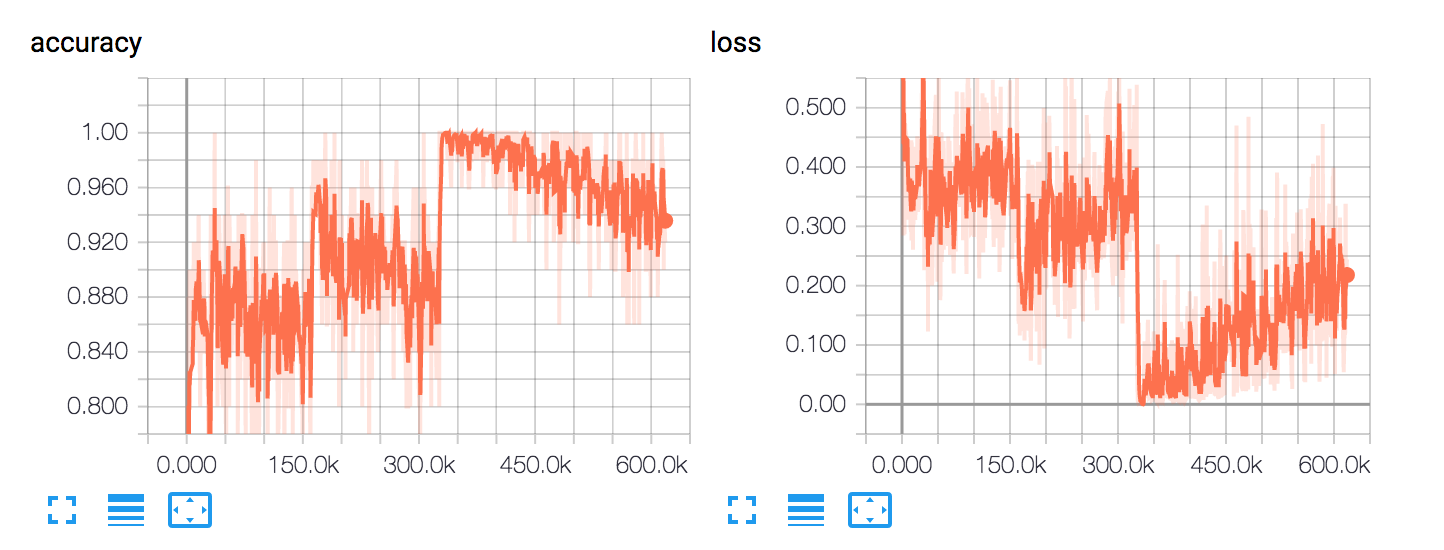
    sess.close()

The model prediction performance is shown as below. The additional layer did not bring a lot of improvements in terms of accuracy.



I added the training accuracy and training loss information in tensorbaord. Interestingly, the accuracy of the model actually decreases after the first half of training data. I am concerned with potential over fitting in the model. In order to study if there are any overfitting, I plot out the training/testing data accuracy and loss comparison.





## Two Hidden Layers Model with Dropout

Even though the accuracy plot does not show any bent in the testing data accuracy, the loss function, after feed more than half of the data, start to show a slightly up trend. I further checked literatures and books to see if there are any techniques I can apply to reduce overfitting and improve prediction power on testing and validation set.

Once technique introduced in Google’s tutorial “Learn TensorFlow and deep learning, without a Ph.D.” is the dropout function. The dropout function randomly select neurons to dropout at each layer to avoid overfitting. One important variable input for the model is the keeping probability. Again, I searched literatures. It seems that the dropout rate of 0.5 brings most variation to the model and can effectively change the model structure. I also used 0.5 as default keep probability in my model. The code is pretty similar to the previous version except that dropout option is added with keep\_prob default to 0.5.

def construct\_neural\_layer\_with\_dropout(input\_x, layer\_nn,name,drop\_out=True,keep\_prob=0.5,activation\_fnc=None):

    with tf.name\_scope(name):

        n\_var=int(input\_x.get\_shape()[1])

        #create random weights

        stddev=2/np.sqrt(n\_var)

        w=tf.Variable(tf.truncated\_normal((n\_var,layer\_nn),stddev=stddev),name="weight",dtype=tf.float32)

        b=tf.Variable(tf.zeros([layer\_nn]),name="biases",dtype=tf.float32)

        output=tf.add(tf.matmul(input\_x,w),b)

        if activation\_fnc=="sigmoid":

            ans= tf.sigmoid(output)

        elif activation\_fnc=="relu":

            ans= tf.nn.relu(output)

        elif activation\_fnc=="softmax":

            ans= tf.nn.softmax(output)

        else:

            ans= output

        if drop\_out:

            return tf.nn.dropout(ans,keep\_prob)

        return ans

This function is an extension to the original one by incorporating dropout feature inside. After the activation function, the result will optionally passing through dropout function to kill certain portion of the neurons.

The network is constructed with below code.

layer1\_nn=10

layer2\_nn=5

input\_x=tf.placeholder(tf.float32,shape=(None,8),name="X")

input\_y=tf.placeholder(tf.int32,shape=(None),name="y")

batch\_size=50

learning\_rate=0.001

hidden1=construct\_neural\_layer\_with\_dropout(input\_x,layer1\_nn,"Hidden\_Layer1",activation\_fnc="relu")

hidden2=construct\_neural\_layer\_with\_dropout(hidden1,layer2\_nn,"Hidden\_Layer2",activation\_fnc="relu")

output=construct\_neural\_layer\_with\_dropout(hidden2,2,"Output\_Layer",drop\_out=False)

loss=tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=input\_y,logits=output))

accuracy=tf.reduce\_mean(tf.cast(tf.nn.in\_top\_k(output,input\_y,1),tf.float32))

tf.summary.scalar('loss', loss)

tf.summary.scalar('accuracy', accuracy)

merged = tf.summary.merge\_all()

train\_writer = tf.summary.FileWriter('NN2\_dropout')

with tf.name\_scope("train"):

    optimizer = tf.train.AdamOptimizer(learning\_rate)

    training\_op = optimizer.minimize(loss)

init = tf.global\_variables\_initializer()

with tf.Session() as sess:

    init.run()

    for iters in range(len(scaled\_train\_set.index.values)//batch\_size):

        X=scaled\_train\_set.iloc[(iters\*batch\_size):((iters+1)\*batch\_size),1:9]

        Y=scaled\_train\_set.iloc[(iters\*batch\_size):((iters+1)\*batch\_size),0]

        opt,train\_loss,train\_accuracy,train\_summ=sess.run([training\_op,loss,accuracy,merged],feed\_dict={input\_x:X,input\_y:Y})

        train\_writer.add\_summary(train\_summ,iters\*batch\_size)

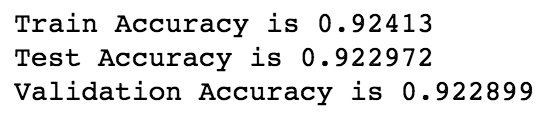
    acc\_train=accuracy.eval(feed\_dict={input\_x:scaled\_train\_set.iloc[:,1:9],input\_y:scaled\_train\_set.iloc[:,0]})

    acc\_test=accuracy.eval(feed\_dict={input\_x:scaled\_test\_set.iloc[:,1:9],input\_y:scaled\_test\_set.iloc[:,0]})

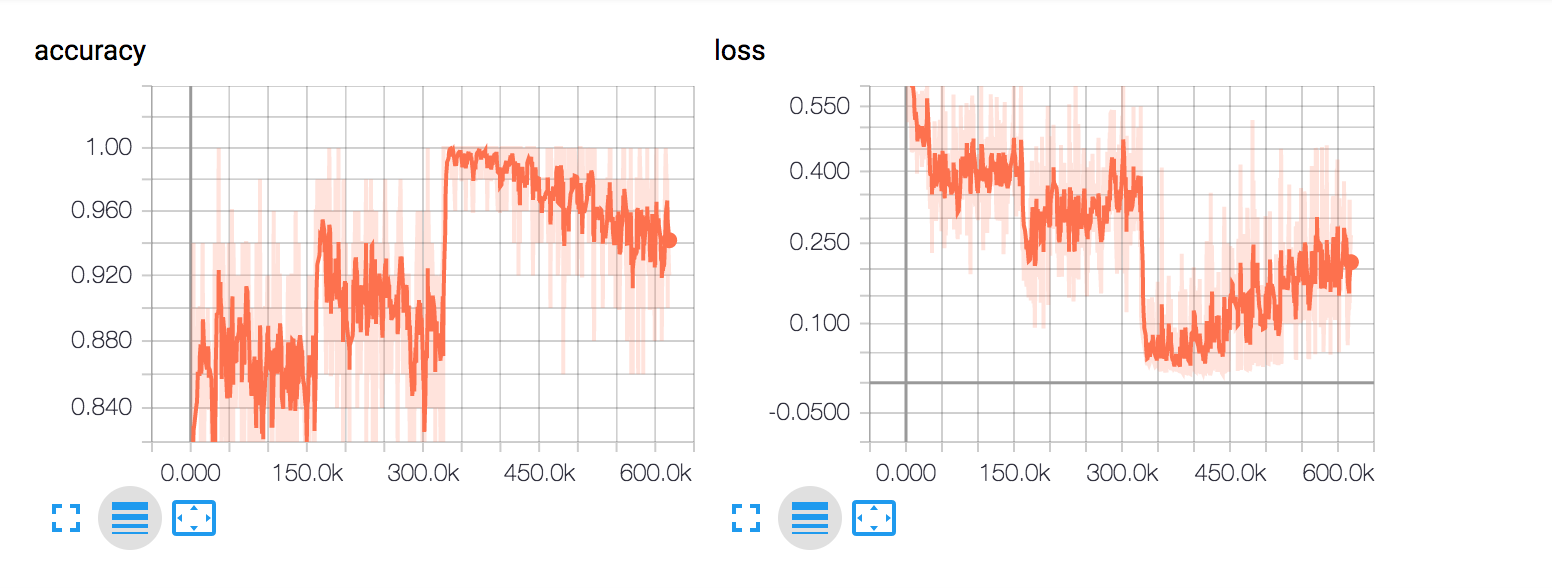
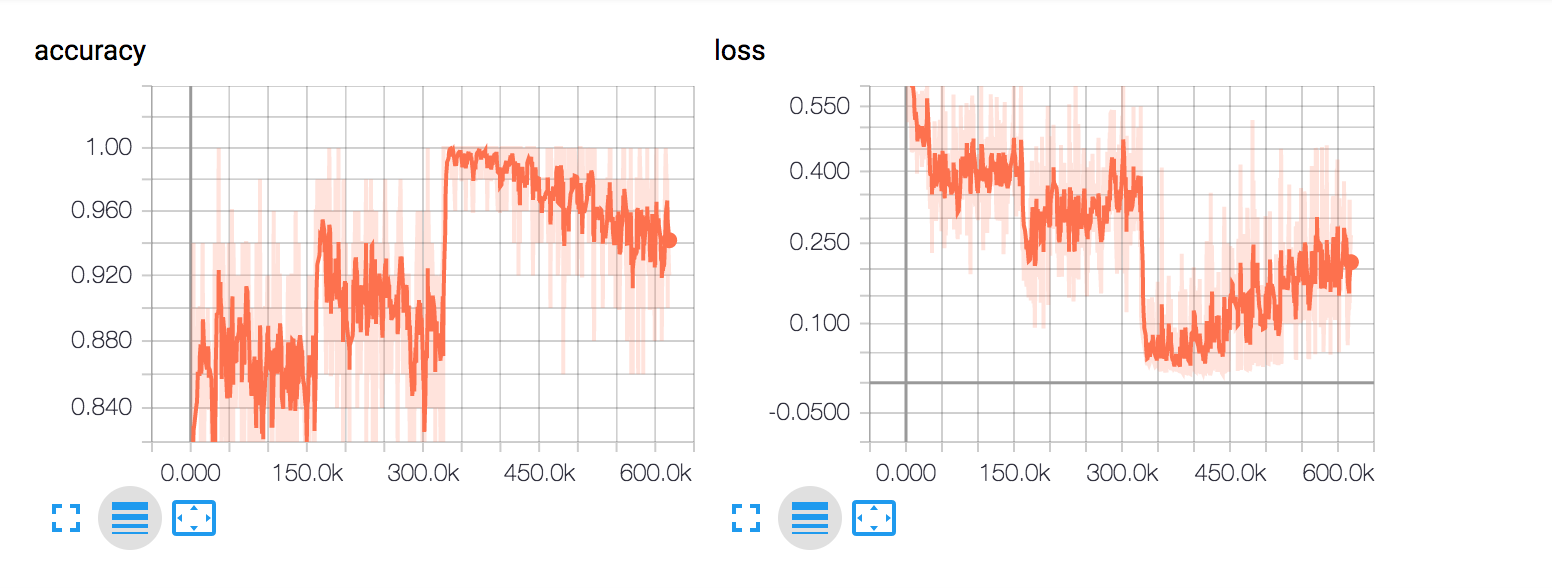
    acc\_validate=accuracy.eval(feed\_dict={input\_x:scaled\_validation\_set.iloc[:,1:9],input\_y:scaled\_validation\_set.iloc[:,0]})

    sess.close()

The result is shown below. Even though we applied dropout function, the model accuracy does not change significantly in he test and validation set.



The plots from tensorboard shows a similar pattern. But the testing set loss function is much smoother in this case. I am not sure what’s the reason behind but I feel the optimizer has more robust result with the dropout function.



## Three Hidden Layers Model with Dropout

Out of curiosity, I further tried to use the same function to construct a three hidden layers model. Because the layer construction function is exactly the same, I will not copy one more time here. The neural network is constructed and trained using the below code.

layer1\_nn=10

layer2\_nn=10

layer3\_nn=5

input\_x=tf.placeholder(tf.float32,shape=(None,8),name="X")

input\_y=tf.placeholder(tf.int32,shape=(None),name="y")

batch\_size=50

learning\_rate=0.001

hidden1=construct\_neural\_layer\_with\_dropout(input\_x,layer1\_nn,"Hidden\_Layer1",keep\_prob=0.5,activation\_fnc="relu")

hidden2=construct\_neural\_layer\_with\_dropout(hidden1,layer2\_nn,"Hidden\_Layer2",keep\_prob=0.5,activation\_fnc="relu")

hidden3=construct\_neural\_layer\_with\_dropout(hidden2,layer3\_nn,"Hidden\_Layer3",drop\_out=False,activation\_fnc="relu")

output=construct\_neural\_layer\_with\_dropout(hidden3,2,"Output\_Layer",drop\_out=False)

loss=tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=input\_y,logits=output))

accuracy=tf.reduce\_mean(tf.cast(tf.nn.in\_top\_k(output,input\_y,1),tf.float32))

tf.summary.scalar('loss', loss)

tf.summary.scalar('accuracy', accuracy)

merged = tf.summary.merge\_all()

train\_writer = tf.summary.FileWriter('NN3\_dropout')

with tf.name\_scope("train"):

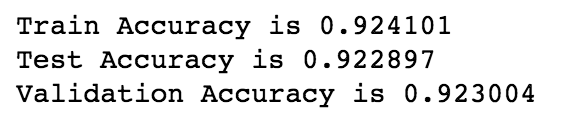
    optimizer = tf.train.AdamOptimizer(learning\_rate)

    training\_op = optimizer.minimize(loss)

init = tf.global\_variables\_initializer()

Here, the two bottom hidden layers both have 10 neurons. It is because I need to apply the dropout function, which will greatly reduce the neurons. Dropping 50% of the neurons at the second hidden layer with too small number of neurons to start easily cost under fitting of the model. The testing set accuracy quickly dropped to around 80% when I set the second hideen layer neurons to 8.

However, such structure does not improve the model prediction significantly. The accuracy on test and validation set still stays around 0.923.



It seems that the uptrend in the loss is no longer there applying 3-hidden layer with dropout enabled. However, the model accuracy does not improve a lot.

## 

## Conclusion and Potential Future Study

After applying various structure on the scaled data, it seems that the neural network indeed has strong prediction power if trained properly. However, the increase of the model may not be easier achieved by adjusting the number of neurons or adding layers. More advanced model need to be studied in order to gain further prediction power. For example, a key factor for the default is the timing and the macro economic environment. Such relationship may not be well-captured in the simple fully connected model. RNN may be a powerful structure in this case.

On the other hand, the study is only restricted to the 8 out of 46 variables available. The left-over data should also have value that have not been explored.

## YouTube Links:

Two minute (short): https://youtu.be/lq8o6mh1WNc

15 minutes (long): https://youtu.be/nxN74MNMMzg

Reference:

*Aurélien Géron,Hands-on Machine Learning with Scikit-Learn and TensorFlow, 2017, ed.1,Chapter 10 & 11*