Project topic:TensorFlow and Neural Networks Applications in HealthCareÿ

Student Name: Liangliang Zhang

Problem Statement:

Use TensorFlow to build a Dense Neural Network that will be used to automatically classify fetal cardiotocogram to different fetal state (N, S, P) based on their diagnostic features data provided by the UCI Machine Learning Repository.

Overview of Technology

TensorFlow and TensorBoard was used to built Multilayer Dense Neural Network model and monitor loss function on training dataset.

TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. (https://www.tensorflow.org/ (https://www.tensorflow.org/))

Descreption of Data

2126 fetal cardiotocograms (CTGs) were automatically processed and the respective diagnostic features measured. The CTGs were also classified by three expert obstetricians and a consensus classification label assigned to each of them. Classification was both with respect to a morphologic pattern (A, B, C. ...) and to a fetal state (N=normal; S=suspect; P=pathologic).

URL: http://archive.ics.uci.edu/ml/machine-learning-databases/00193/ (http://archive.ics.uci.edu/ml/machine-learning-databases/00193/)

Size: 1.66 MB., sample size: 2130

Format of data file: .xls file of Microsoft Excel

Hardware

Windows PC with Intel Core M-5Y10c CPU (0.8GHz, 998MHz) and 4GB RAM

Sofeware

Anaconda with Python 3.6.1

TensorFlow 1.3.0 https://pypi.python.org/pypi/tensorflow/1.3.0 (https://pypi.python.org/pypi/tensorflow/1.3.0)

Lessons learned & Pros/Cons

After tuning, my final Neural Network model gives a prediction accuracy of ~92% in training data and a prediction accuracy of ~90% in validation data. This model performs reasonably well and I suppose that if we have more observations, especially observations of the minority class, we could have built a more powerful neural network.

YouTube URLs:

short (2 min): https://www.youtube.com/watch?v=dwUjhR7LHFY (https://www.youtube.com/watch?v=dwUjhR7LHFY)

long (15 min): https://www.youtube.com/watch?v=25v_I7LKyBU (https://www.youtube.com/watch?v=25v_I7LKyBU)

In [1]:
```
# import libraries
import tensorflow as tf
import numpy as np
import pandas as pd
```

Steps & Demonstration

1. load data

Data fiel CTG.xls was downloaded from http://archive.ics.uci.edu/ml/machine-learning-databases/00193/ (http://archive.ics.uci.edu/ml/machine-learning-databases/00193/)

In [2]:
```
#load data
ctg = pd.read_excel('CTG.xls', sheetname = 2)
```

2. Data cleaning

In [3]:
```
#check data shape
print ('CTG data shape:,', ctg.shape)
#check data head
ctg.head()
```

CTG data shape:, (2130, 40)

Out[3]:

	FileName	Date	SegFile	b	e	LBE	LB	AC	FM	UC	...	C	D
0	NaN	NaT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
1	Variab10.txt	1996-12-01	CTG0001.txt	240.0	357.0	120.0	120.0	0.0	0.0	0.0	...	0.0	0.0
2	Fmcs_1.txt	1996-05-03	CTG0002.txt	5.0	632.0	132.0	132.0	4.0	0.0	4.0	...	0.0	0.0
3	Fmcs_1.txt	1996-05-03	CTG0003.txt	177.0	779.0	133.0	133.0	2.0	0.0	5.0	...	0.0	0.0
4	Fmcs_1.txt	1996-05-03	CTG0004.txt	411.0	1192.0	134.0	134.0	2.0	0.0	6.0	...	0.0	0.0

5 rows × 40 columns

In [4]:
```
#check data tail
ctg.tail()
```

Out[4]:

	FileName	Date	SegFile	b	e	LBE	LB	AC	FM	UC	...	C
2125	S8001045.dsp	1998-06-06	CTG2127.txt	1576.0	3049.0	140.0	140.0	1.0	0.0	9.0	...	0.0
2126	S8001045.dsp	1998-06-06	CTG2128.txt	2796.0	3415.0	142.0	142.0	1.0	1.0	5.0	...	0.0
2127	NaN	NaT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
2128	NaN	NaT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
2129	NaN	NaT	NaN	NaN	NaN	NaN	NaN	NaN	564.0	23.0	...	NaN

5 rows × 40 columns

Drop the column of Filename, Date and SegFile, as these information has absolutely no predictive power for determing the state of a CTG image. Keeping these information will only confuse our model when training neural network.

In [5]:
```
ctg.drop(['FileName', 'Date', 'SegFile'], axis = 1, inplace = True)
```

Drop first row as it is blank, then drop a few rows from the bottom as they contain meaningless information.

In [6]:
```
ctg_clear = ctg.drop(ctg.index[[0, 2127, 2128, 2129]])
```

Check if there are any missing values.

In [7]:
```
print ('Having missing values? :', ctg_clear.isnull().any().any())
```

```
Having missing values? : False
```

Let's check the shape and statistical descriptions after cleaning:

In [8]:
```
print ('Data shape after cleaning', ctg_clear.shape)
```

```
Data shape after cleaning (2126, 37)
```

In [33]:
```
ctg_clear.head()
```

Out[33]:

	b	e	LBE	LB	AC	FM	UC	ASTV	MSTV	ALTV	...	C	D	E	AD	DE	LD
1	240.0	357.0	120.0	120.0	0.0	0.0	0.0	73.0	0.5	43.0	...	0.0	0.0	0.0	0.0	0.0	0.0
2	5.0	632.0	132.0	132.0	4.0	0.0	4.0	17.0	2.1	0.0	...	0.0	0.0	0.0	1.0	0.0	0.0
3	177.0	779.0	133.0	133.0	2.0	0.0	5.0	16.0	2.1	0.0	...	0.0	0.0	0.0	1.0	0.0	0.0
4	411.0	1192.0	134.0	134.0	2.0	0.0	6.0	16.0	2.4	0.0	...	0.0	0.0	0.0	1.0	0.0	0.0
5	533.0	1147.0	132.0	132.0	4.0	0.0	5.0	16.0	2.4	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 37 columns

In [10]:
```
ctg_clear.describe()
```

Out[10]:

	b	e	LBE	LB	AC	FM	UC
count	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000	2126.000000
mean	878.439793	1702.877234	133.303857	133.303857	2.722484	7.241298	3.659925
std	894.084748	930.919143	9.840844	9.840844	3.560850	37.125309	2.847094
min	0.000000	287.000000	106.000000	106.000000	0.000000	0.000000	0.000000
25%	55.000000	1009.000000	126.000000	126.000000	0.000000	0.000000	1.000000
50%	538.000000	1241.000000	133.000000	133.000000	1.000000	0.000000	3.000000
75%	1521.000000	2434.750000	140.000000	140.000000	4.000000	2.000000	5.000000
max	3296.000000	3599.000000	160.000000	160.000000	26.000000	564.000000	23.000000

8 rows × 37 columns

3. Extract feature and lables

The dataset has two types of labels: morphologic pattern and fetal state. In this project, we only use fetal state label to perform a 3-class classification. Then fetal labels was then onehot encoded to dummy variables.

In [34]:
```
features = ctg_clear.iloc[:, :-12].values
labels = ctg_clear.iloc[:, -1].values

labels_onehot = pd.get_dummies(labels)

print ('Number of abservations:', features.shape[0])
print ('Number of features:', features.shape[1])
print ('number of labels:', labels_onehot.shape[1])
```

```
Number of abservations: 2126
Number of features: 25
number of labels: 3
```

4. Train and validation data split

Then entire dataset was randomly split into training (80% 1700 cases) and validation (20% 426 cases) dataset. Train dataset is used for training our neural network, and validation datasetis used for testing the accurarcy of our model.

In [184]:
```
from sklearn.model_selection import train_test_split
# Take 1/5 images from the training data, and leave the remainder in training
train_dataset, valid_dataset, train_labels, valid_labels = train_test_split(featu
print('Training data/label shape: ', train_dataset.shape, train_labels.shape)
print('Validation data/label shape: ', valid_dataset.shape, valid_labels.shape)
```

```
Training data/label shape:  (1700, 25) (1700, 3)
Validation data/label shape:  (426, 25) (426, 3)
```

In [185]:
```
#check the propotion of each class in train and validation data
print ('Propotion for each class in train data:', np.sum(train_labels, axis=0)/tr
print ('Propotion for each class in validaion data:', np.sum(valid_labels, axis=0
```

```
Propotion for each class in train data: [ 0.77411765  0.14117647  0.08470588]
Propotion for each class in validaion data: [ 0.79577465  0.12910798  0.0751173
7]
```

The propotion for each class is similar in training and validation dataset, so we will have all information needed in training data.

5. Dense Neural Network (DNN) model

5.1 Define a few useful functions

In [186]:
```
# calculate accuracy by identifying validation cases where the model's highest-pr
def accuracy(predictions, labels):
    correct_prediction = tf.equal(tf.argmax(predictions, 1), tf.argmax(labels, 1)
    accuracy_pct = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)) * 100.
    #another way to calculate this is to use np like following
    return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / p
    #return accuracy_pct.eval()
```

In [187]:
```
def weight_variable(shape, name):
    initial = tf.truncated_normal(shape, stddev=1e-4)
    #initial = tf.truncated_normal(shape, stddev=np.sqrt(2.0/shape[0]))
    return tf.Variable(initial, name=name)

def bias_variable(shape, name):
    #initial = tf.constant(0.1, shape=shape)
    initial = tf.zeros(shape)
    return tf.Variable(initial, name)

split_by_half = lambda x,k : int(x/2**k)
```

5.2 Simple 2-layer DNN model with GradientDescentOptimizer

In [188]:
```
valid_dataset = valid_dataset.astype(np.float32)
n_labels = 3
batch_size = 99
flattened_size = train_dataset.shape[1]
hidden_nodes = 100

graph = tf.Graph()
with graph.as_default():

    # Input data.
    tf_train_dataset = tf.placeholder(tf.float32, shape=(batch_size, flattened_si
    tf_train_labelset = tf.placeholder(tf.float32, shape=(batch_size, n_labels), 
    tf_valid_dataset = tf.constant(valid_dataset, name="ValidationData")

    # Variables.
    layer1_weights = tf.Variable(tf.truncated_normal([flattened_size, hidden_node
    layer1_biases = tf.Variable(tf.zeros([hidden_nodes]), name="biases1")
    layer2_weights = tf.Variable(tf.truncated_normal([hidden_nodes, n_labels]), n
    layer2_biases = tf.Variable(tf.ones([n_labels]), name="biases2")

    # Model.
    def model(data, name):
        with tf.name_scope(name) as scope:
            layer1 = tf.add(tf.matmul(data, layer1_weights), layer1_biases, name=
            hidden1 = tf.nn.relu(layer1, name="relu1")
            layer2 = tf.add(tf.matmul(hidden1, layer2_weights), layer2_biases, na
            return layer2

    # Training computation.
    logits = model(tf_train_dataset, name="logits")
    #loss function
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, 

    # Optimizer.
    optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)

    # Predictions for the training, validation
    train_prediction = tf.nn.softmax(logits)
    valid_prediction = tf.nn.softmax(model(tf_valid_dataset, name="validation"))
```

In [189]:
```
# define run model function
def run_session(num_epochs, name):
    with tf.Session(graph=graph) as session:
        tf.global_variables_initializer().run()
        merged = tf.summary.merge_all()
        writer = tf.summary.FileWriter("tmp/tensorflowlogs", session.graph)
        print("Initialized model:", name)
        for epoch in range(num_epochs):
            offset = (epoch * batch_size) % (train_labels.shape[0] - batch_size)
            batch_data = train_dataset[offset:(offset + batch_size), :]
            batch_labels = train_labels[offset:(offset + batch_size), :]
            feed_dict = {tf_train_dataset : batch_data, tf_train_labelset : batch_
            _, l, predictions = session.run([optimizer, loss, train_prediction], 
            if (epoch % 500 == 0):
                print('Minibatch loss at epoch %d: %f' % (epoch, l))
                print('Minibatch accuracy: %.1f%%' % accuracy(predictions, batch_
                print('Validation accuracy: %.1f%%' % accuracy(valid_prediction.e
```

In [179]:
```
run_session(5001, "DNN_2layer")
```

```
Initialized model: DNN_2layer
Minibatch loss at epoch 0: 10123.847656
Minibatch accuracy: 14.1%
Validation accuracy: 75.6%
Minibatch loss at epoch 500: 0.536054
Minibatch accuracy: 84.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 1000: 0.751381
Minibatch accuracy: 73.7%
Validation accuracy: 75.6%
Minibatch loss at epoch 1500: 0.622334
Minibatch accuracy: 79.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 2000: 0.629530
Minibatch accuracy: 80.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 2500: 0.752382
Minibatch accuracy: 74.7%
Validation accuracy: 75.6%
Minibatch loss at epoch 3000: 0.658178
Minibatch accuracy: 78.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 3500: 0.616110
Minibatch accuracy: 79.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 4000: 0.674650
Minibatch accuracy: 76.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 4500: 0.668530
Minibatch accuracy: 76.8%
Validation accuracy: 75.6%
Minibatch loss at epoch 5000: 0.686899
Minibatch accuracy: 76.8%
Validation accuracy: 75.6%
```

After 5000 epoches, both training and validation accuracies are arround 76%, which is similar to a blind guess of first class.

Next, we modify several parameters of our DNN model to see if we can improve model performance. Modifications are listed below:

1. More hidden layers
2. Regularization and dropout to avoid over fitting
3. Altinative optimizer

Also, summary for loss function, train accuracy and validation accuracy were added to TensroBoard, so we can keep tracking our model performance.

5.3 4-layer DNN model with regularization, dropout and AdamOptimizer

In [194]:
```
batch_size = 340
flattened_size = train_dataset.shape[1]
hidden_nodes = 512
lamb_reg = 0.001
learning_rate = 0.001  #  learning rate for the momentum optimizer

graph = tf.Graph()
with graph.as_default():

    # Input data.
    tf_train_dataset = tf.placeholder(tf.float32, shape=(batch_size, flattened_si
    tf_train_labelset = tf.placeholder(tf.float32, shape=(batch_size, n_labels), 
    tf_valid_dataset = tf.constant(valid_dataset, name="ValidationData")
    tf_valid_labelset = tf.constant(valid_labels, name="ValidationLabels")
    # Variables.
    layer1_weights = weight_variable([flattened_size, hidden_nodes], name="weight
    layer1_biases = bias_variable([hidden_nodes], name="biases1")
    layer2_weights = weight_variable([hidden_nodes, split_by_half(hidden_nodes,1)
    layer2_biases = bias_variable([split_by_half(hidden_nodes,1)], name="biases2"
    layer3_weights = weight_variable([split_by_half(hidden_nodes,1), split_by_hal
    layer3_biases = bias_variable([split_by_half(hidden_nodes,2)], name="biases3"
    layer4_weights = weight_variable([split_by_half(hidden_nodes,2), n_labels], n
    layer4_biases = bias_variable([n_labels], name="biases4")

    keep_prob = tf.placeholder("float", name="keep_prob")

    def model(data, name, proba=keep_prob):
        with tf.name_scope(name) as scope:
            layer1 = tf.add(tf.matmul(data, layer1_weights), layer1_biases, name=
            hidden1 = tf.nn.dropout(tf.nn.relu(layer1), proba, name="dropout1")
            layer2 = tf.add(tf.matmul(hidden1, layer2_weights), layer2_biases, na
            hidden2 = tf.nn.dropout(tf.nn.relu(layer2), proba, name="dropout2")
            layer3 = tf.add(tf.matmul(hidden2, layer3_weights), layer3_biases, na
            hidden3 = tf.nn.dropout(tf.nn.relu(layer3), proba)
            layer4 = tf.add(tf.matmul(hidden3, layer4_weights), layer4_biases, na
            return layer4

    # Training computation.
    logits = model(tf_train_dataset, "logits", keep_prob)
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, 
    regularizers = (tf.nn.l2_loss(layer1_weights) + tf.nn.l2_loss(layer1_biases) 
                    tf.nn.l2_loss(layer2_weights) + tf.nn.l2_loss(layer2_biases) 
                    tf.nn.l2_loss(layer3_weights) + tf.nn.l2_loss(layer3_biases) 
                    tf.nn.l2_loss(layer4_weights) + tf.nn.l2_loss(layer4_biases)

    # Add the regularization term to the loss.
    loss += lamb_reg * regularizers
    #loss = tf.reduce_mean(loss + lamb_reg * regularizers)

    # Optimizer
    #global_step = tf.Variable(0, name="globalstep")  # count  number of steps ta
    #optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001, epsilon=1e-04).minimi

    # Predictions for the training, validation, and test data.
    train_prediction = tf.nn.softmax(logits)
```

```
    valid_prediction = tf.nn.softmax(model(tf_valid_dataset, "validation", 1.0))
    #saver = tf.train.Saver()   # a saver variable to save the model

    # acuuracy for training data
    train_correct_prediction = tf.equal(tf.cast(tf.argmax(logits, 1), tf.float32)
    accuracy_train = tf.reduce_mean(tf.cast(train_correct_prediction, tf.float32)
    # acuuracy for validation data
    valid_correct_prediction = tf.equal(tf.cast(tf.argmax(model(tf_valid_dataset,
    accuracy_valid = tf.reduce_mean(tf.cast(valid_correct_prediction, tf.float32)
```

In [195]:

```
def run_session_2(num_epochs, name, k_prob=1.0):

    with tf.Session(graph=graph) as session:
        tf.global_variables_initializer().run()

        # summaries
        loss_summary = tf.summary.scalar('Loss', loss)

        train_accuracy_summary = tf.summary.scalar('train_accuracy', accuracy_tra
        valid_accuracy_summary = tf.summary.scalar('valid_accuracy', accuracy_val

        merged = tf.summary.merge_all()
        writer = tf.summary.FileWriter("tmp/tensorflowlogs_3", session.graph)

        print('Initialized model:', name,"\n")
        for epoch in range(num_epochs):
            offset = (epoch * batch_size) % (train_labels.shape[0] - batch_size)
            batch_data = train_dataset[offset:(offset + batch_size), :]
            batch_labels = train_labels[offset:(offset + batch_size), :]
            feed_dict = {tf_train_dataset : batch_data, tf_train_labelset : batch
            _, l, predictions = session.run([optimizer, loss, train_prediction], 
            writer.add_summary(loss_summary.eval(feed_dict=feed_dict), epoch)
            writer.add_summary(train_accuracy_summary.eval(feed_dict=feed_dict), 
            writer.add_summary(valid_accuracy_summary.eval(feed_dict=feed_dict), 
            #writer.add_summary(learning_rate_summary.eval(), epoch)
            if (epoch % 500 == 0):
                print("Minibatch loss at epoch {}: {}".format(epoch, l))
                print("Minibatch accuracy: {:.1f}".format(accuracy(predictions, b
                print("Validation accuracy: {:.1f}\n".format(accuracy(valid_predi
        #save_path = saver.save(session, "tmp/" + name +".ckpt")
        #print("Model saved in file: %s" % save_path)
```

In [196]:
```
run_session_2(5001, "DNN_4layer_Adam", 1.0)
```

```
Initialized model: DNN_4layer_Adam

Minibatch loss at epoch 0: 1.098612666130066
Minibatch accuracy: 80.6
Validation accuracy: 79.6

Minibatch loss at epoch 500: 0.22571192681789398
Minibatch accuracy: 92.4
Validation accuracy: 89.0

Minibatch loss at epoch 1000: 0.17441688477993011
Minibatch accuracy: 94.1
Validation accuracy: 87.1

Minibatch loss at epoch 1500: 0.13863994181156158
Minibatch accuracy: 95.6
Validation accuracy: 88.0

Minibatch loss at epoch 2000: 0.06984758377075195
Minibatch accuracy: 98.5
Validation accuracy: 88.3

Minibatch loss at epoch 2500: 0.1138841062784195
Minibatch accuracy: 97.1
Validation accuracy: 88.3

Minibatch loss at epoch 3000: 0.04885120317339897
Minibatch accuracy: 99.7
Validation accuracy: 88.7

Minibatch loss at epoch 3500: 0.04376523569226265
Minibatch accuracy: 99.4
Validation accuracy: 89.9

Minibatch loss at epoch 4000: 0.05714946240186691
Minibatch accuracy: 99.1
Validation accuracy: 89.9

Minibatch loss at epoch 4500: 0.0718587189912796
Minibatch accuracy: 98.2
Validation accuracy: 88.7

Minibatch loss at epoch 5000: 0.035477880388498306
Minibatch accuracy: 99.7
Validation accuracy: 90.8
```

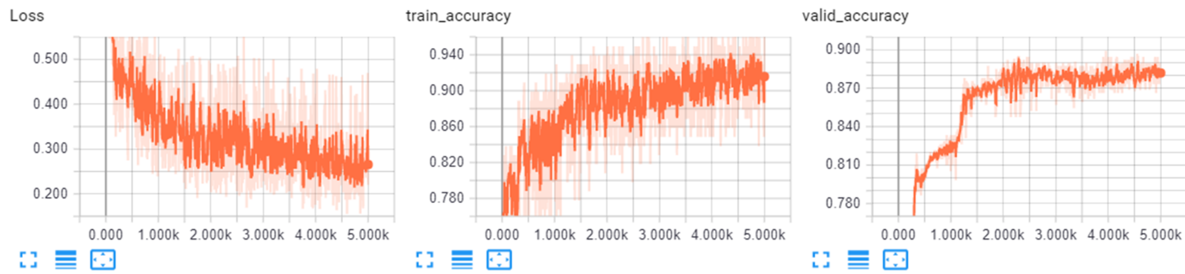
In [145]:
```
run_session_2(5001, "DNN_4layer_Adam", 1.0)
```

```
Initialized model: DNN_4layer_Adam

Minibatch loss at epoch 0: 1.098612666130066
Minibatch accuracy: 81.0
Validation accuracy: 76.8

Minibatch loss at epoch 500: 0.3719613552093506
Minibatch accuracy: 88.0
Validation accuracy: 84.5

Minibatch loss at epoch 1000: 0.2221064567565918
Minibatch accuracy: 89.0
Validation accuracy: 87.1

Minibatch loss at epoch 1500: 0.24829697608947754
Minibatch accuracy: 88.0
Validation accuracy: 87.6

Minibatch loss at epoch 2000: 0.28749698400497437
Minibatch accuracy: 88.0
Validation accuracy: 88.3

Minibatch loss at epoch 2500: 0.2003413736820221
Minibatch accuracy: 94.0
Validation accuracy: 86.9

Minibatch loss at epoch 3000: 0.10720705986022949
Minibatch accuracy: 97.0
Validation accuracy: 89.2

Minibatch loss at epoch 3500: 0.17392706871032715
Minibatch accuracy: 94.0
Validation accuracy: 90.4

Minibatch loss at epoch 4000: 0.19757121801376343
Minibatch accuracy: 91.0
Validation accuracy: 89.0

Minibatch loss at epoch 4500: 0.12272512167692184
Minibatch accuracy: 97.0
Validation accuracy: 90.6

Minibatch loss at epoch 5000: 0.08677855879068375
Minibatch accuracy: 99.0
Validation accuracy: 89.4
```

Modified Neural Network model gives a prediction accuracy of ~99% in training data and a prediction accuracy of ~90% in validation data. Visualization from TensroBoard is shown below:

In [1]:
```
from IPython.display import Image
Image("TensorBoard.png")
```

Out[1]:



Different optimizer (MomentumOptimizer, AdamOptimizer, GradientDescentOptimizer), learning rate (0.0001, 0.001, 0.01, 0.1) and keep probability (1.0, 0.8, 0.5) were tested. The final DNN model (AdamOptimizer, learning rate =0.001, keep probability=1.0), which has the best performance on validation data, was shown above.

6. Conclusion

Our final Neural Network model gives a prediction accuracy of ~92% in training data and a prediction accuracy of ~90% in validation data. This model performs reasonably well and I suppose that if we have more observations, especially observations of the minority class, we could have built a more powerful neural network.