Assignment - Final Project **Aaron Liu 2019/04/20**

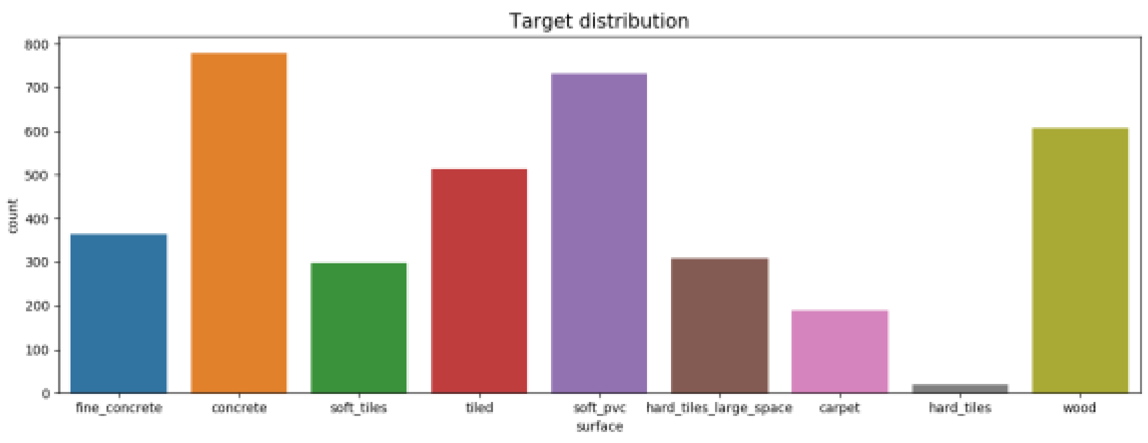
Objective:

I am trying to do multi-classification of the Inertial Measurement Units (IMU sensors) data collecting from small mobile robots, which contains nine categories of different surface, including concrete, fine concrete, soft tiles, hard tiles and so on (I show the 9 categories in the graph below) by using multiple ANN Classifiers and make efforts to experiment with preprocessing the input data, feature engineering, customizing loss function, tuning hyper parameter, finding optimal network structure and regularization.

That is, I would use the ANN classifiers to help small mobile robots recognize the floor surface they're standing on. By fully understanding the environment, robots could properly navigate tasks and won't fail on the job.

The source URL(s) for the data and description is in the link:

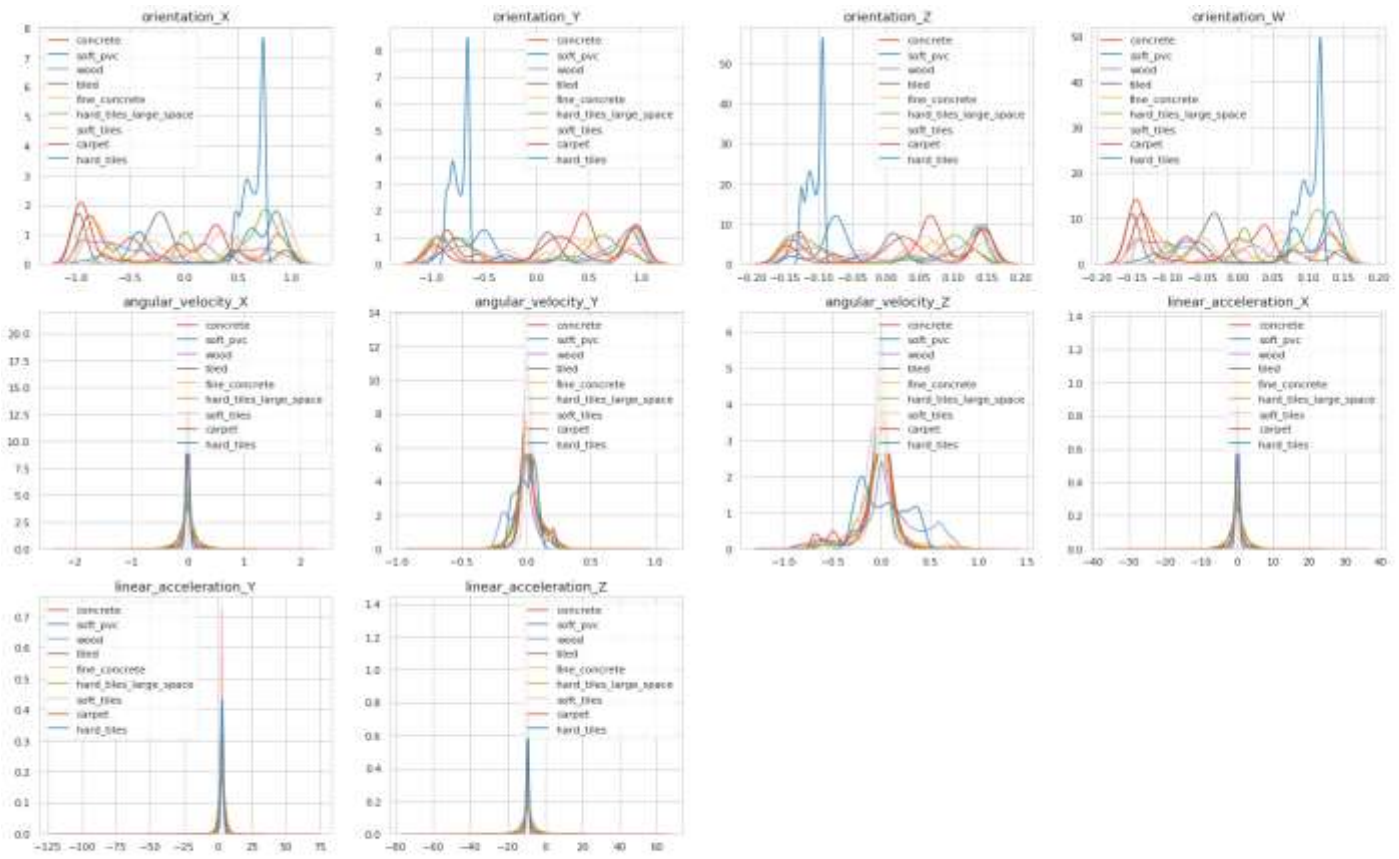
https://www.kaggle.com/c/career-con-2019/data



Characteristics and brief description of the data set:

The training set includes ten variables collecting from sensors, including orientation X, orientation Y, orientation Z, orientation W, angular velocity X, angular velocity Y, angular velocity Z, linear acceleration X, linear acceleration Y and linear acceleration Z and three ID columns including row id, series id and measurement number. The orientation variables encode the current angles how the robot is oriented as a quaternion. Angular velocity describes the angle and the speed of motion, and linear acceleration describes how the speed is changing at different times. For the output part, I need to predict and generate the right surface of each series id for the test set.

The distribution of the nine categories among 10 given attributes is in the following chart. From the chart, I find that the distributions among orientation_X, orientation_Y, orientation_Z and orientation_W are really difference from other attributes, especially for hard tiles category, so I decide to do some feature engineering with orientation_X, orientation_Y, orientation_Z and orientation_W in the data preprocessing part.



Data preprocessing steps:

1. Normalize some given attributes, including orientation_X, orientation_Y, orientation_Z and orientation_W.
2. Calculate euler angles from given attributes which have been normalized in the first step (orientation_X, orientation_Y, orientation_Z, orientation_W). This step is important because the robot navigate their directions on different surface mainly by euler angles.
3. Generate more features by using the given attributes. Since that it is a classification problem, I think it would be useful and practical to use the mean, maximum, minimum, standard deviation and range of the given attributes to help my ANN classifier understand the data more.
4. Since each series id includes 128 measurements and also the measurements in the same series id are recorded on the same surface, I group 487680 measurements by 3810 series ids. Therefore, there would be 3810 observations after this step.
5. Replace the NaN and infinity value with 0.
6. Save the data in csv form for future use after doing feature engineering and load the data.
7. Scale the loaded data for fitting neural network model.
8. Transform my target (surfaces) from string form to numeric form for classification
9. Split the data into training set and test set (80/20 proportion)
10. Split the training data into training set and validation set
11. Encode the target (surfaces), vectorize the labels and the data is ready for doing classification.

Description of the input variables:

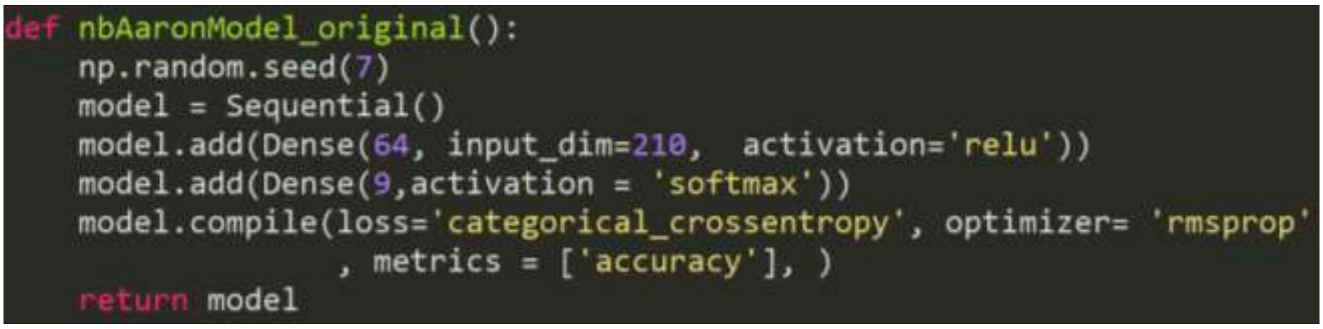
After preprocessing the data, I have 2048 observations with 210 attributes in my training set, 1000 observations with 210 attributes in my validation set and 762 observations with 210 attributes in my test set. Accordingly, I make the input dimension to 210 in my model.

Experiments before reaching my final model:

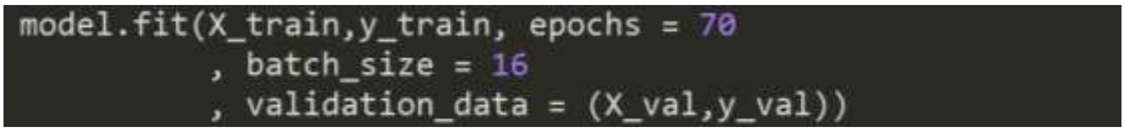
For my very first model, I use a simple network structure that I have only two layers in my model, including 64 nodes, 210 input dimension, and “relu” activation function for the input layer and 9 output units and “softmax” activation function for the output layer. For the training algorithm, I use “rmsprop” as the optimizer and “categorical_crossentropy” as the loss function.

Running time of the original model: 5 seconds

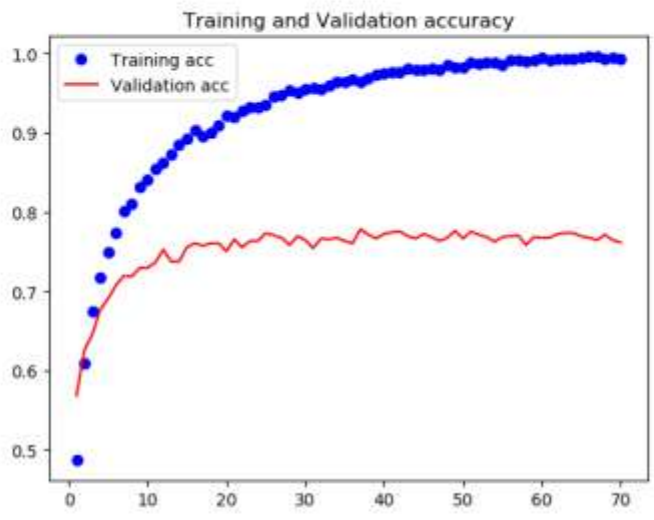
The test accuracy of the original model is 79.66%



```python
def nbAaronModel_original():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(64, input_dim=210,  activation='relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss='categorical_crossentropy', optimizer= 'rmsprop'
                  , metrics = ['accuracy'], )
    return model
```



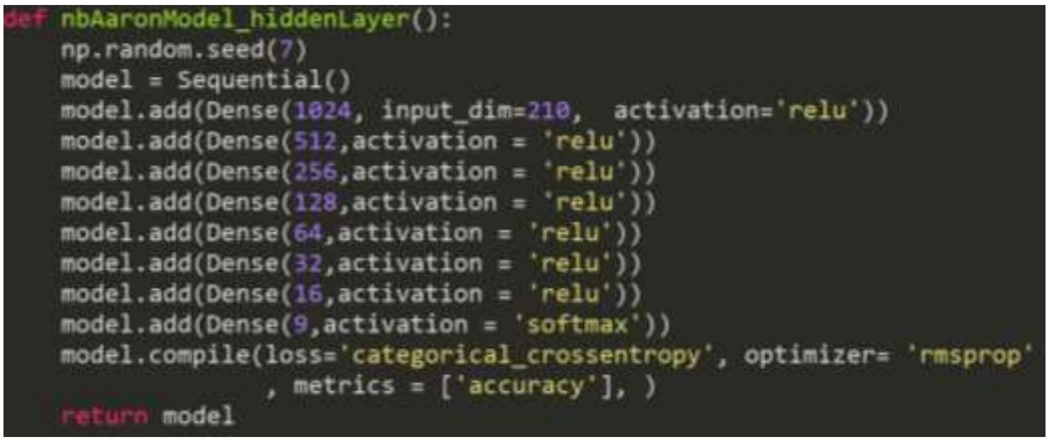
```python
model.fit(X_train,y_train, epochs = 70
          , batch_size = 16
          , validation_data = (X_val,y_val))
```



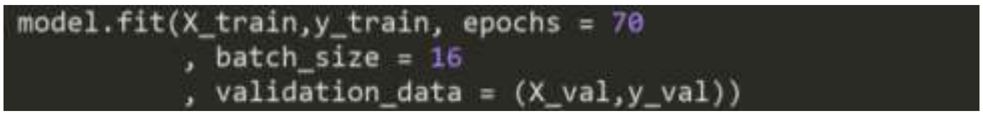
First experiment: Add hidden layers

For the first experiment, I add 6 more hidden layers based on try and error. And the nodes for each layer are 1024, 512, 256, 128, 64, 32, 16 and 9. The activation function for input layer and hidden layers are all "relu" and that for output layer is "softmax". The training algorithm remain the same.

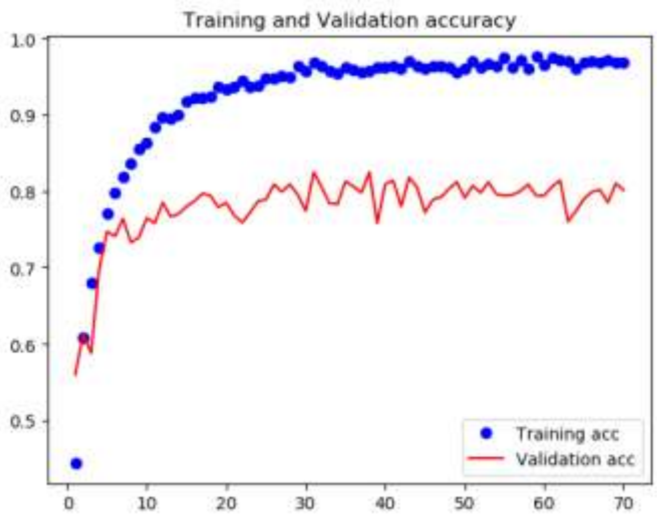
Running time of the model with more hidden layers: 2 minutes and 35 seconds
The test accuracy of this model is 83.20%



```
def nbAaronModel_hiddenLayer():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(1024, input_dim=210,  activation='relu'))
    model.add(Dense(512,activation = 'relu'))
    model.add(Dense(256,activation = 'relu'))
    model.add(Dense(128,activation = 'relu'))
    model.add(Dense(64,activation = 'relu'))
    model.add(Dense(32,activation = 'relu'))
    model.add(Dense(16,activation = 'relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss='categorical_crossentropy', optimizer= 'rmsprop'
                  , metrics = ['accuracy'], )
    return model
```



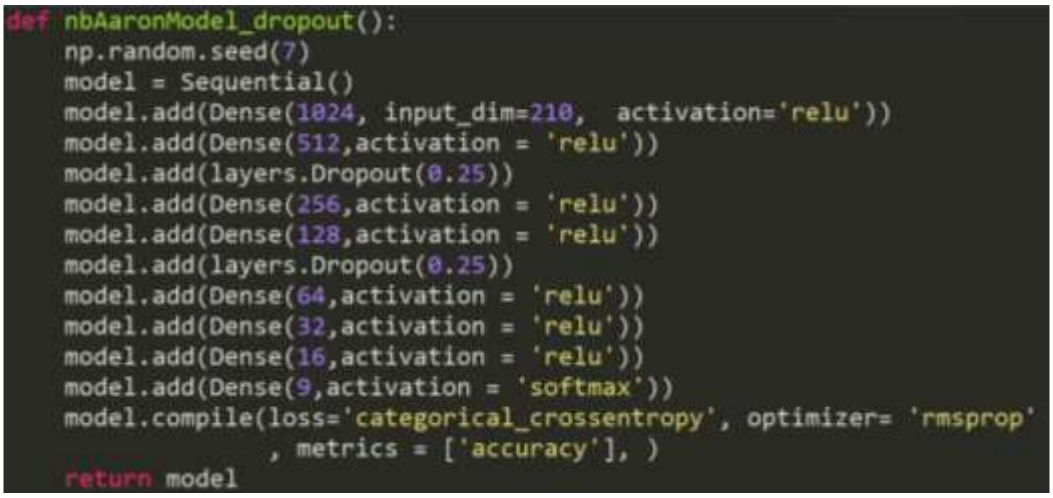
```
model.fit(X_train,y_train, epochs = 70
          , batch_size = 16
          , validation_data = (X_val,y_val))
```



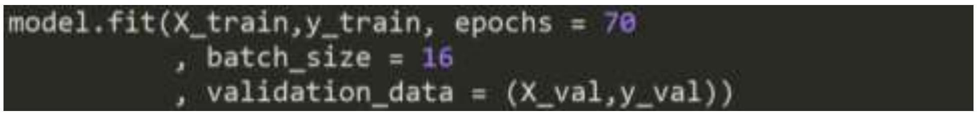
Second experiment: Regularization with drop out layers

Since that the test accuracy increase from 79.66% to 83.20% after first experiment, I use the same network structure and add drop out layers after two of hidden layers to see if I could drop out some features that are not so important and make my classification more accurate. After training, I find that the test accuracy does not improve, I choose not to add drop out layers in my final model.

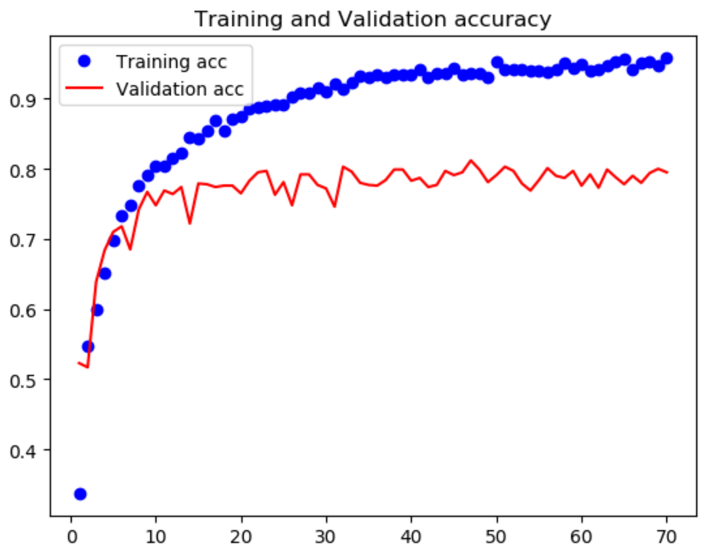
Running time of the model with drop out layers: 2 minutes and 45 seconds
The test accuracy of this model is 81.76%



```
def nbAaronModel_dropout():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(1024, input_dim=210,  activation='relu'))
    model.add(Dense(512,activation = 'relu'))
    model.add(layers.Dropout(0.25))
    model.add(Dense(256,activation = 'relu'))
    model.add(Dense(128,activation = 'relu'))
    model.add(layers.Dropout(0.25))
    model.add(Dense(64,activation = 'relu'))
    model.add(Dense(32,activation = 'relu'))
    model.add(Dense(16,activation = 'relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss='categorical_crossentropy', optimizer= 'rmsprop'
                  , metrics = ['accuracy'], )
    return model
```

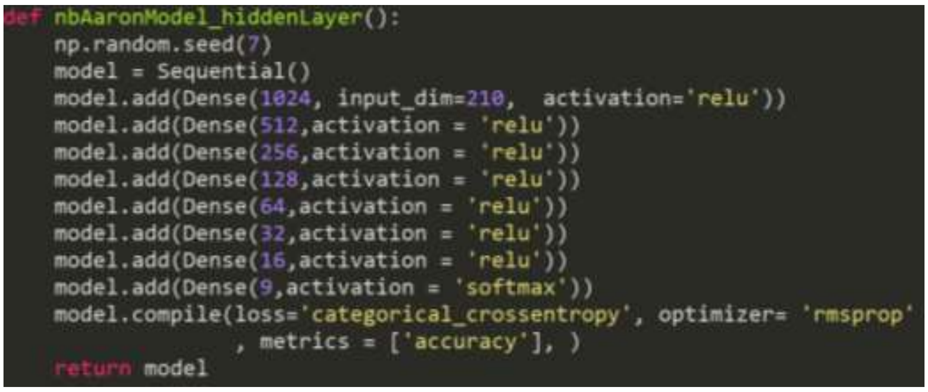


```
model.fit(X_train,y_train, epochs = 70
          , batch_size = 16
          , validation_data = (X_val,y_val))
```



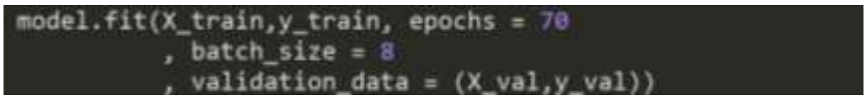
Third experiment: Change batch size

For this experiment, I try to figure out if I could make improvement on the accuracy by increasing or decreasing the batch size. First, I decrease the batch size to 8 when fitting the model from experiment 1. Then, I experiment with increasing the batch size to 40 when fitting the model from experiment 1. For the model training with larger batch size, the test accuracy increase from 83.2% (experiment 1) to 84.51%.

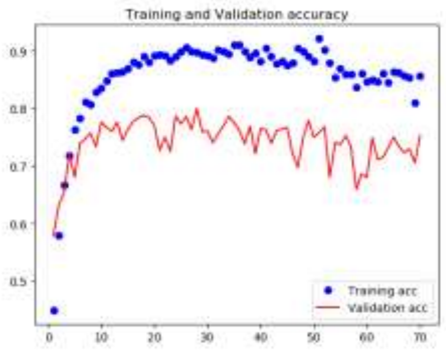


```
def nbAaronModel_hiddenLayer():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(1024, input_dim=210,  activation='relu'))
    model.add(Dense(512,activation = 'relu'))
    model.add(Dense(256,activation = 'relu'))
    model.add(Dense(128,activation = 'relu'))
    model.add(Dense(64,activation = 'relu'))
    model.add(Dense(32,activation = 'relu'))
    model.add(Dense(16,activation = 'relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss='categorical_crossentropy', optimizer= 'rmsprop'
                  , metrics = ['accuracy'], )
    return model
```

Batch size = 8:



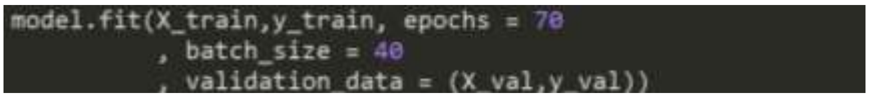
```
model.fit(X_train,y_train, epochs = 70
           , batch_size = 8
           , validation_data = (X_val,y_val))
```



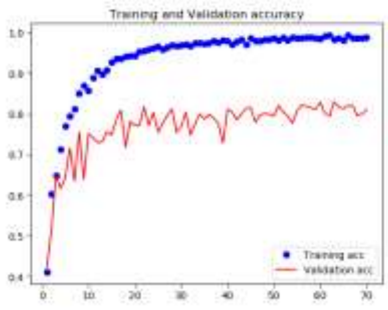
Running time of the model from experiment 1 with smaller batch size: 4 minutes and 42 seconds

The test accuracy of this model is 72.83%

Batch size = 40:



```
model.fit(X_train,y_train, epochs = 70
           , batch_size = 40
           , validation_data = (X_val,y_val))
```



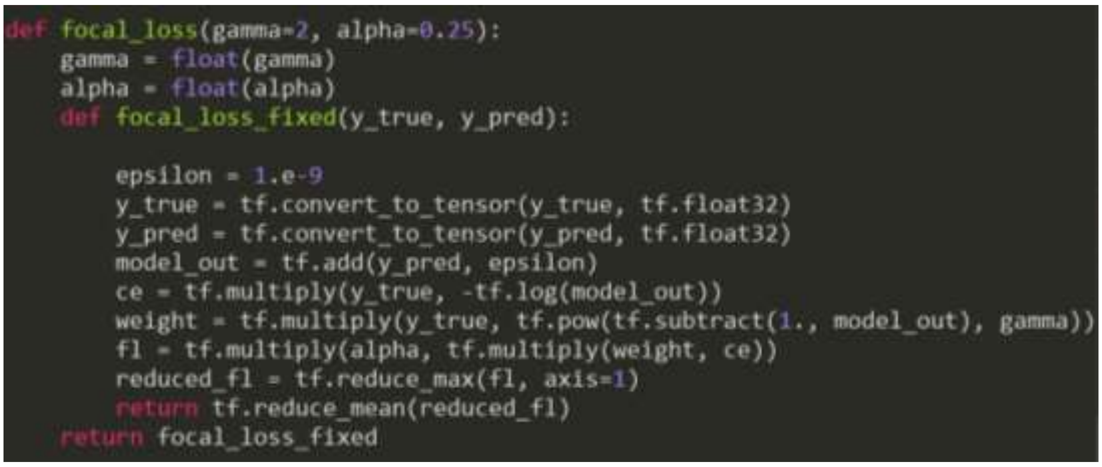
Running time of the model from experiment 1 with larger batch size: 1 minutes and 15 seconds

The test accuracy of this model is 84.51%

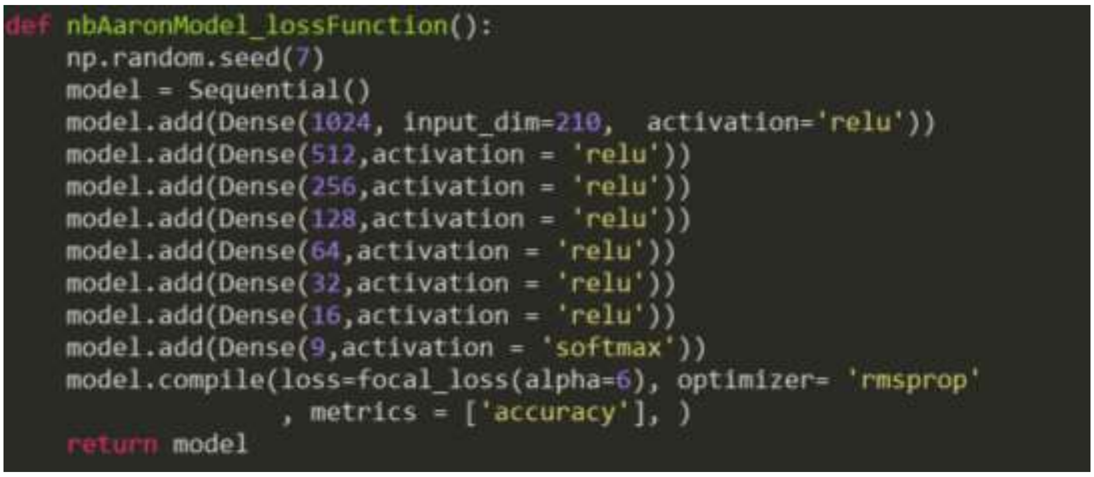
Forth experiment: Customize loss function for imbalanced data

The targets (surfaces) in the data set are imbalanced, for example; there are 793 observations of concrete but only 21 observations of hard tiles. I think the imbalanced data might influence my classification so I customize the loss function to deal with it. The test accuracy do increase from 84.51% to 85.22%.

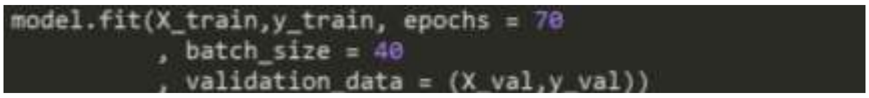
Running time of the model with customized loss function: 1 minutes and 17 seconds
The test accuracy of this model is 85.22%



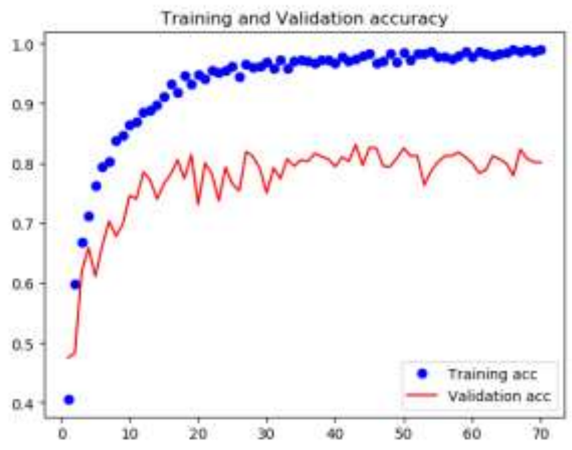
```python
def focal_loss(gamma=2, alpha=0.25):
    gamma = float(gamma)
    alpha = float(alpha)
    def focal_loss_fixed(y_true, y_pred):

        epsilon = 1.e-9
        y_true = tf.convert_to_tensor(y_true, tf.float32)
        y_pred = tf.convert_to_tensor(y_pred, tf.float32)
        model_out = tf.add(y_pred, epsilon)
        ce = tf.multiply(y_true, -tf.log(model_out))
        weight = tf.multiply(y_true, tf.pow(tf.subtract(1., model_out), gamma))
        fl = tf.multiply(alpha, tf.multiply(weight, ce))
        reduced_fl = tf.reduce_max(fl, axis=1)
        return tf.reduce_mean(reduced_fl)
    return focal_loss_fixed
```



```python
def nbAaronModel_lossFunction():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(1024, input_dim=210,  activation='relu'))
    model.add(Dense(512,activation = 'relu'))
    model.add(Dense(256,activation = 'relu'))
    model.add(Dense(128,activation = 'relu'))
    model.add(Dense(64,activation = 'relu'))
    model.add(Dense(32,activation = 'relu'))
    model.add(Dense(16,activation = 'relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss=focal_loss(alpha=6), optimizer= 'rmsprop'
                  , metrics = ['accuracy'], )
    return model
```



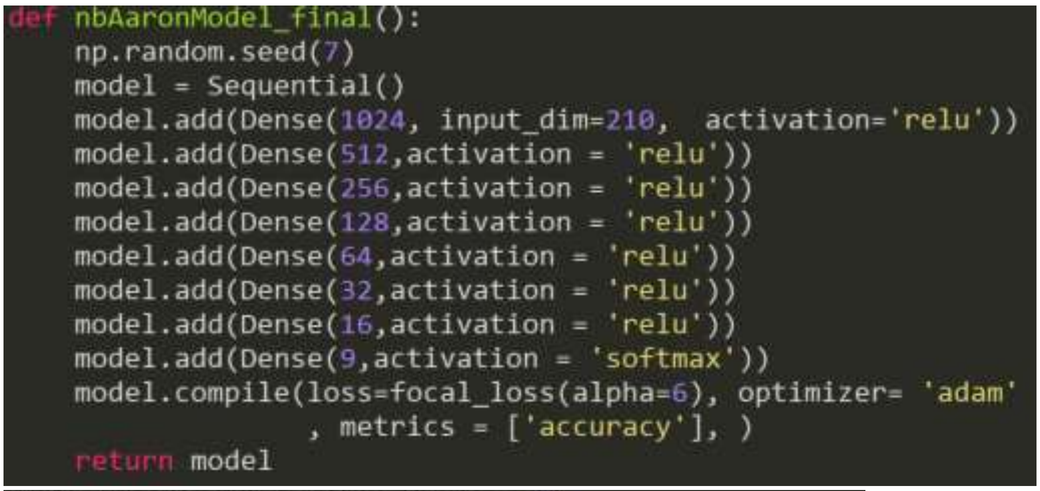
```python
model.fit(X_train,y_train, epochs = 70
          , batch_size = 40
          , validation_data = (X_val,y_val))
```



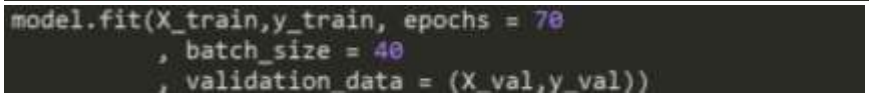
Fifth experiment: Change different optimizers

For the previous few experiments, I use “rmsprop” as the optimizer of my models, and I want to change it to “adam” and obverse the result. After training, the test accuracy increase again from 85.22% to 85.93% so I decide to use “adam” as the optimizer in my final model.

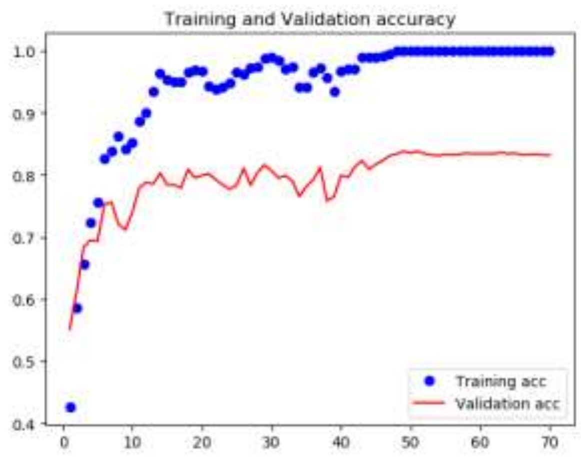
Running time of the model with optimizer “adam”: 1 minutes and 12 seconds
The test accuracy of this model is 85.93%



```
def nbAaronModel_final():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(1024, input_dim=210,  activation='relu'))
    model.add(Dense(512,activation = 'relu'))
    model.add(Dense(256,activation = 'relu'))
    model.add(Dense(128,activation = 'relu'))
    model.add(Dense(64,activation = 'relu'))
    model.add(Dense(32,activation = 'relu'))
    model.add(Dense(16,activation = 'relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss=focal_loss(alpha=6), optimizer= 'adam'
                  , metrics = ['accuracy'], )
    return model
```

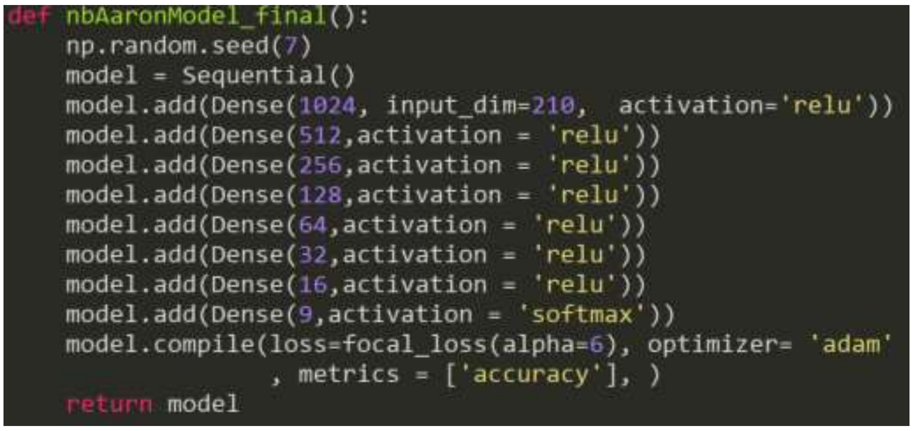


```
model.fit(X_train,y_train, epochs = 70
          , batch_size = 40
          , validation_data = (X_val,y_val))
```

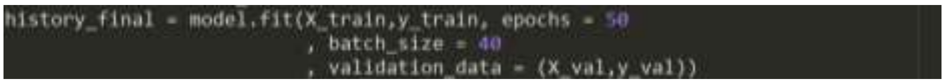


Sixth experiment: Change epochs

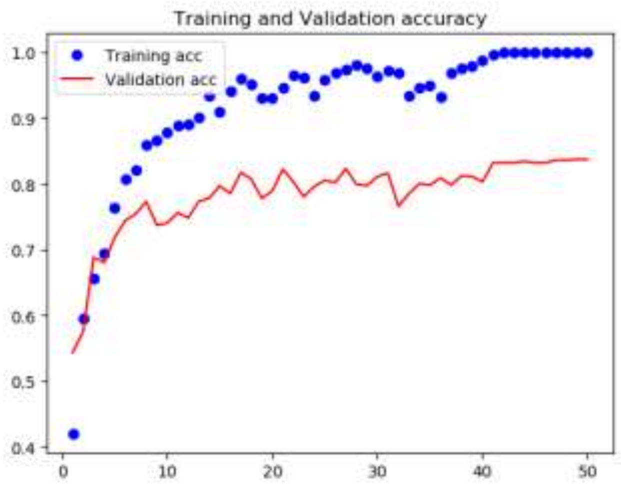
After doing several experiments and generating my final model, I find that the accuracy stop increasing and remain in a steady state from the graph in my previous experiment. As a result, I decide to decrease the epoch to 50 when training my final model. The test accuracy also remain the same as the previous experiment.



```
def nbAaronModel_final():
    np.random.seed(7)
    model = Sequential()
    model.add(Dense(1024, input_dim=210,  activation='relu'))
    model.add(Dense(512,activation = 'relu'))
    model.add(Dense(256,activation = 'relu'))
    model.add(Dense(128,activation = 'relu'))
    model.add(Dense(64,activation = 'relu'))
    model.add(Dense(32,activation = 'relu'))
    model.add(Dense(16,activation = 'relu'))
    model.add(Dense(9,activation = 'softmax'))
    model.compile(loss=focal_loss(alpha=6), optimizer= 'adam'
                  , metrics = ['accuracy'], )
    return model
```



```
history_final = model.fit(X_train,y_train, epochs = 50
                                , batch_size = 40
                                , validation_data = (X_val,y_val))
```



Running time of the model from with less epochs (epoch = 50): 52 seconds
The test accuracy of this model is 85.95%

Final model ,training algorithm and metrics:

My final classification model is a 8-layer backpropagation neural network, which includes 1024, 512, 256, 128, 64, 32, 16, and 9 nodes in each layer respectively. In the input layer and each hidden layer, I use “relu” as the activation function. Since that it’s a problem of multi-classification with 9 categories, I use “softmax” as the activation function and put 9 output units for the output layer, choose “adam” as the optimizer and customize loss function to deal with the imbalanced data. Last but not least, I pick “accuracy” as my metrics to evaluate the performance of my ANN classifier. I don’t tune learning rate and momentum in my “adam” optimizer because I find it not work well for this dataset.

Method to validate my ANN classifier:

I split the training set further into train set and validation set and add validation set in the model.fit function to validate the performance (accuracy) of my final model.

Results:

After doing feature engineering and experimenting with more hidden layers, customized loss function, larger batch size and less epochs, the test accuracy of my final model is 85.93%. Hope the result could help robots navigate their direction and task more accurately and do better jobs.