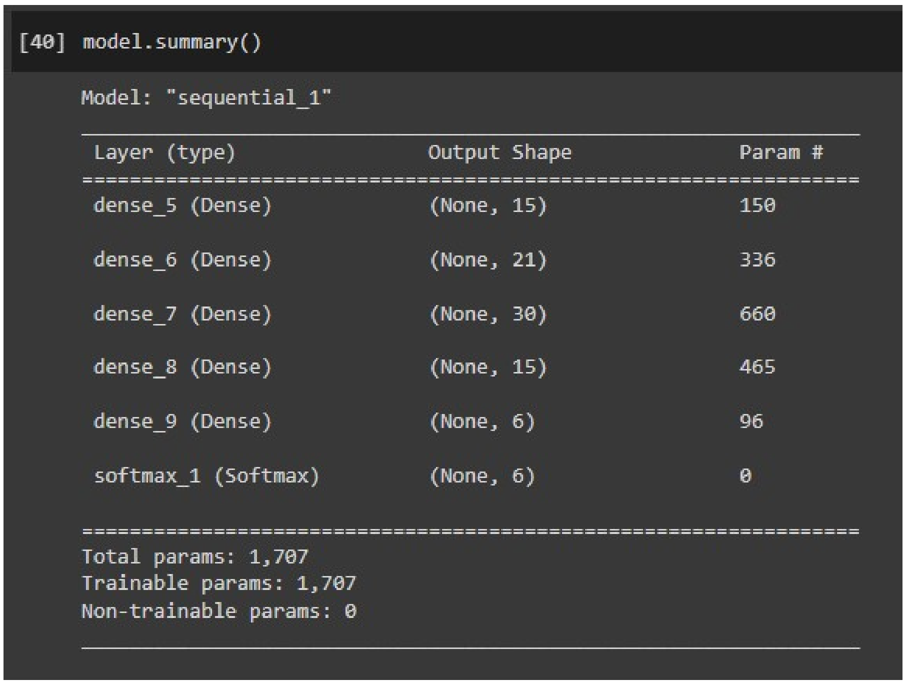
Report Homework 2

A diagram of your network and the parameters used (such as size, no. of neurons, learning rate, etc.)



```
[40] model.summary()

    Model: "sequential_1"
    _________________________________________________________________
     Layer (type)                Output Shape              Param #
    =================================================================
     dense_5 (Dense)             (None, 15)                150

     dense_6 (Dense)             (None, 21)                336

     dense_7 (Dense)             (None, 30)                660

     dense_8 (Dense)             (None, 15)                465

     dense_9 (Dense)             (None, 6)                 96

     softmax_1 (Softmax)         (None, 6)                 0

    =================================================================
    Total params: 1,707
    Trainable params: 1,707
    Non-trainable params: 0
    _________________________________________________________________
```

Input layer: 6

1st Hidden layer: 15

2nd Hidden layer: 21

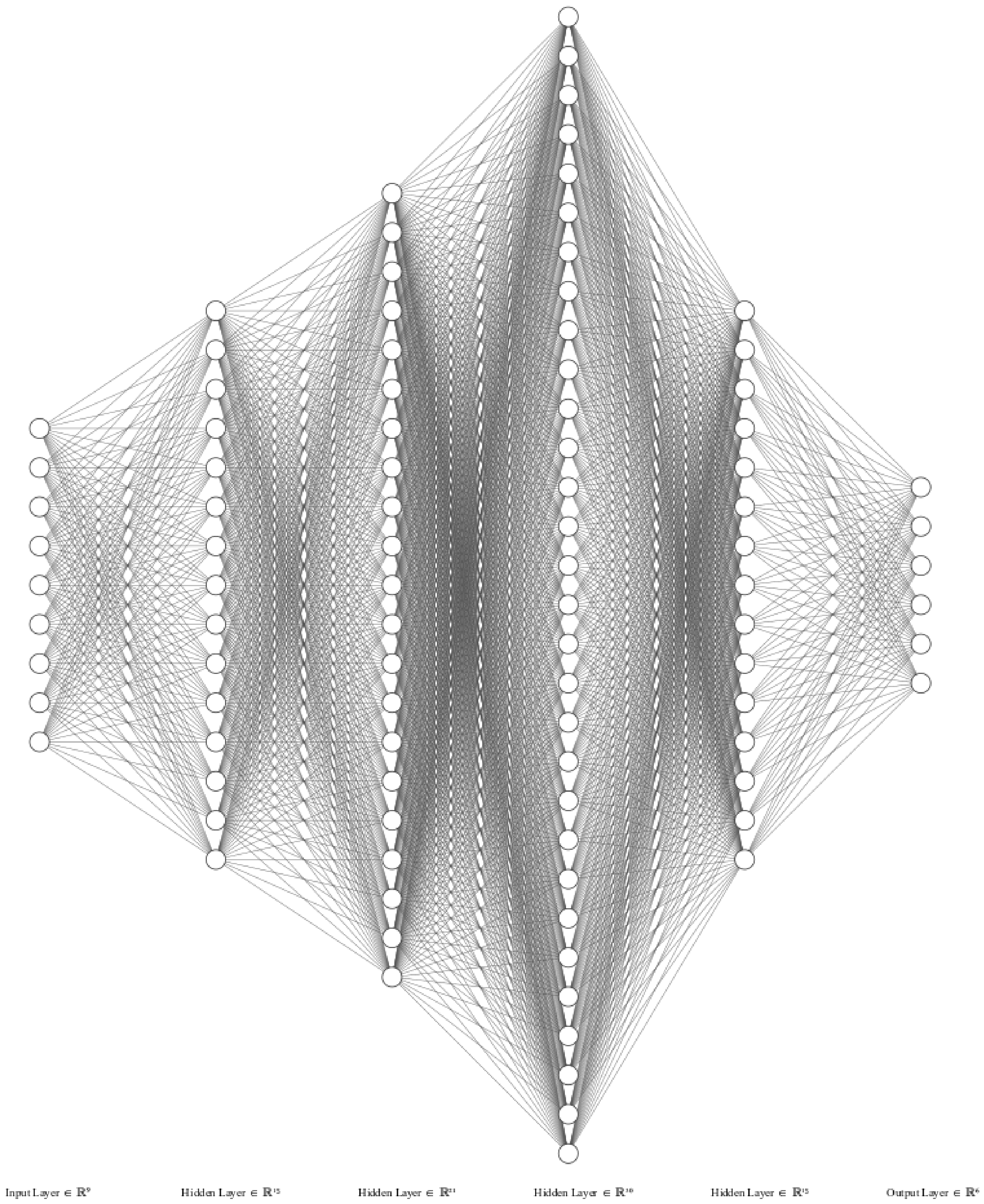
3rd Hidden layer: 30

4th Hidden layer: 15

5th Hidden layer: 6

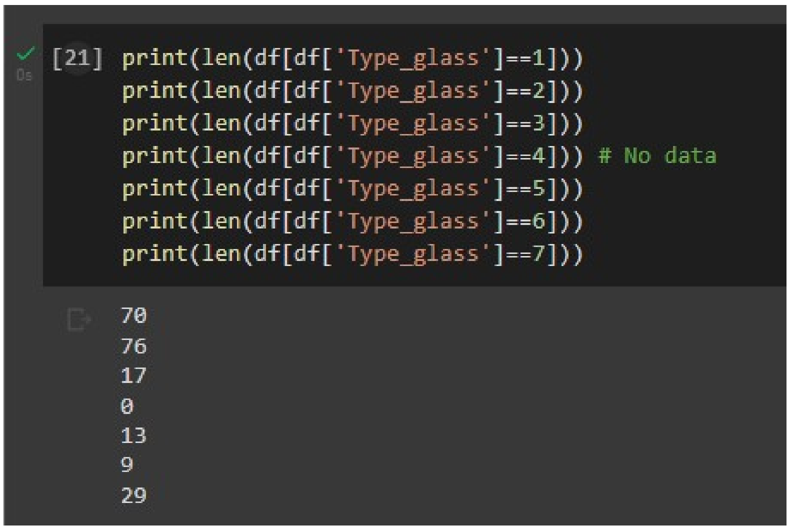
Then, Softmax function.

The neural network diagram next page is generated using https://alexlenail.me/NN-SVG/


Input Layer $\in \mathbb{R}^9$
Hidden Layer $\in \mathbb{R}^{15}$
Hidden Layer $\in \mathbb{R}^{21}$
Hidden Layer $\in \mathbb{R}^{30}$
Hidden Layer $\in \mathbb{R}^{15}$
Output Layer $\in \mathbb{R}^6$

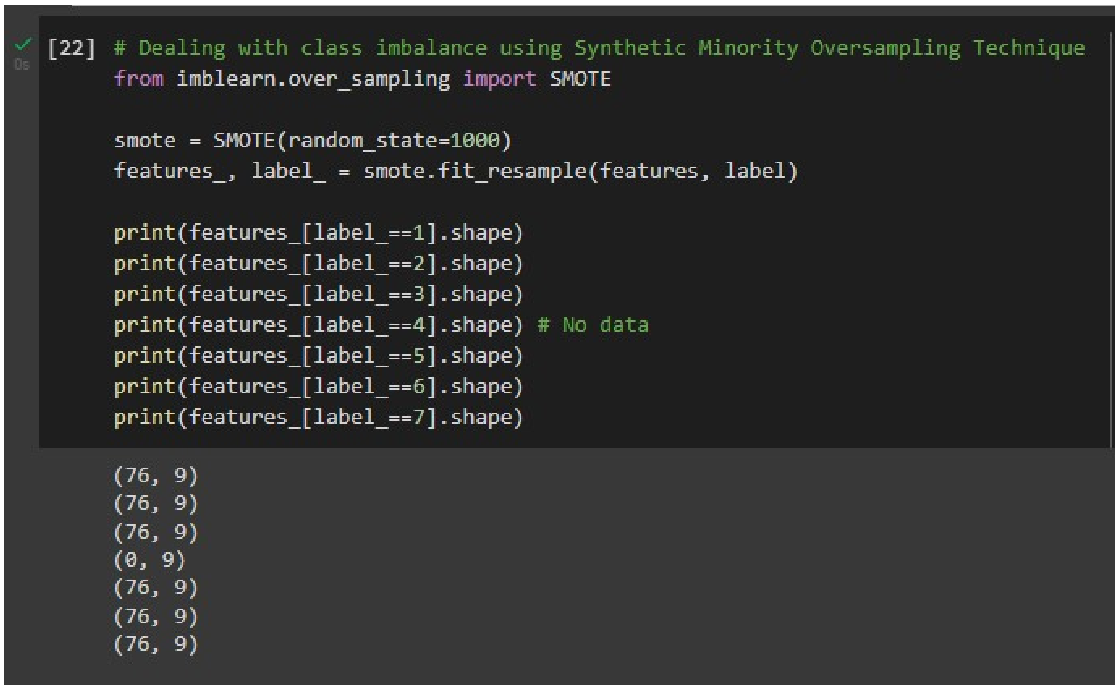
Explanation how you normalize the data and perform class balancing (by code or by hand)

A class balancing should be performed before normalizing data.



```
[21] print(len(df[df['Type_glass']==1]))
     print(len(df[df['Type_glass']==2]))
     print(len(df[df['Type_glass']==3]))
     print(len(df[df['Type_glass']==4])) # No data
     print(len(df[df['Type_glass']==5]))
     print(len(df[df['Type_glass']==6]))
     print(len(df[df['Type_glass']==7]))

     70
     76
     17
     0
     13
     9
     29
```

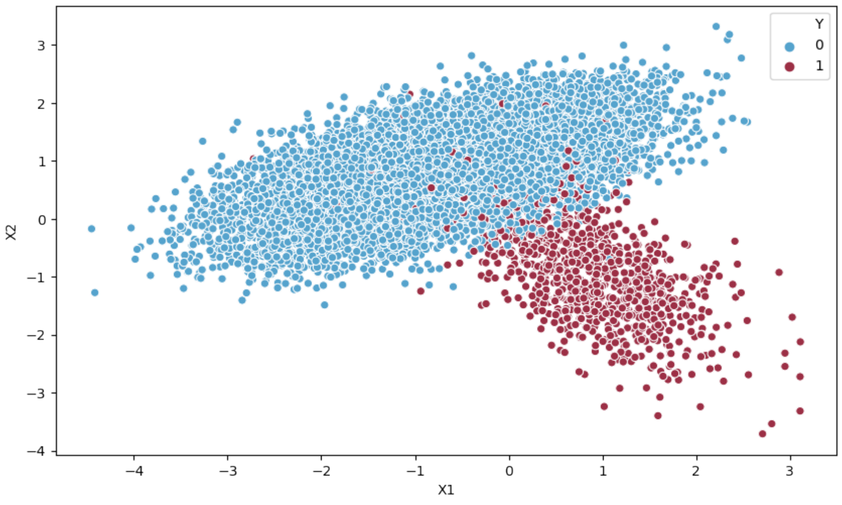
First, I investigated on how many samples in each type of label. It is clear that there are class imbalance, the first and second label are a lot more than the rest (there was no data for label 4).



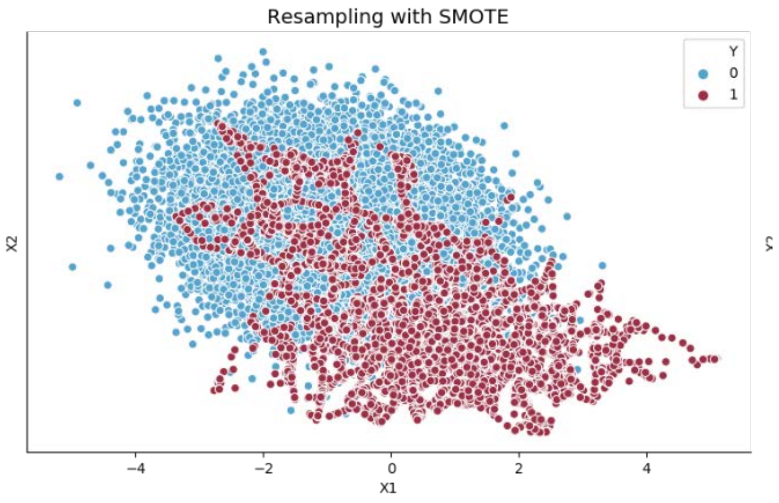
```
[22] # Dealing with class imbalance using Synthetic Minority Oversampling Technique
     from imblearn.over_sampling import SMOTE

     smote = SMOTE(random_state=1000)
     features_, label_ = smote.fit_resample(features, label)

     print(features_[label_==1].shape)
     print(features_[label_==2].shape)
     print(features_[label_==3].shape)
     print(features_[label_==4].shape) # No data
     print(features_[label_==5].shape)
     print(features_[label_==6].shape)
     print(features_[label_==7].shape)

     (76, 9)
     (76, 9)
     (76, 9)
     (0, 9)
     (76, 9)
     (76, 9)
     (76, 9)
```

Working with imbalanced datasets presents a challenge because most machine learning techniques ignore the minority class, which results in poor performance even though this performance is typically the most crucial.

Oversampling the minority class is one way to deal with unbalanced datasets. Duplicating examples from the minority class is the simplest method, but these examples don't provide any new insight into the model. Instead, new examples can be created by synthesizing the existing ones. The Synthetic Minority Oversampling Technique, or SMOTE for short, is a type of data augmentation for the minority class.

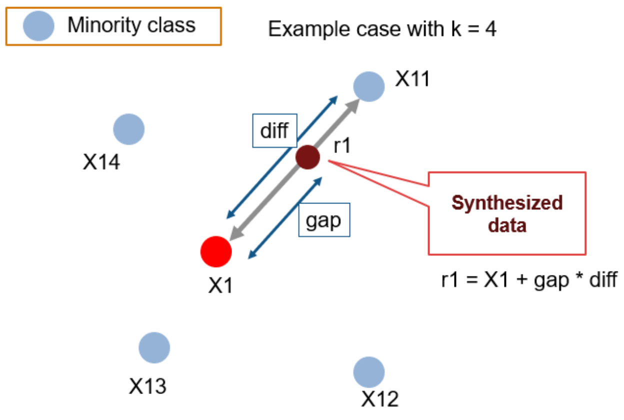


Example of majority (0) and minority (1) class

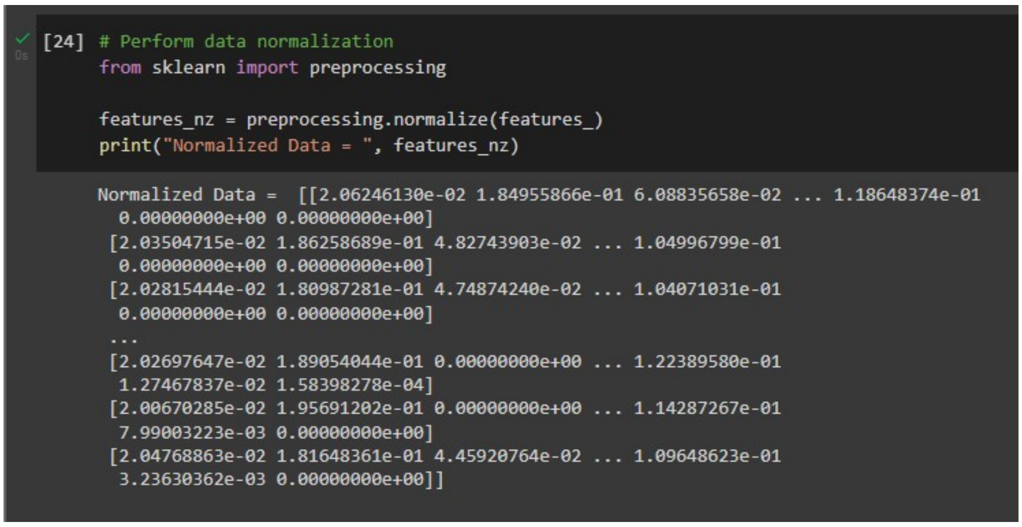


After applying SMOTE resampling

The total number of oversampling observations, N, is first established. The binary class distribution is typically chosen to be 1:1. But, if necessary, that could be turned down. The iteration then begins by first picking a random member of a positive class. The KNNs for that instance are then obtained (there are 5 by default). Finally, N is picked from among these K instances to interpolate new synthetic instances. To do that, the distance between the feature vector and its neighbors is calculated using any distance metric. The previous feature vector is now multiplied by this difference plus any random value within the range of [0, 1]. Below is a visual representation of this:

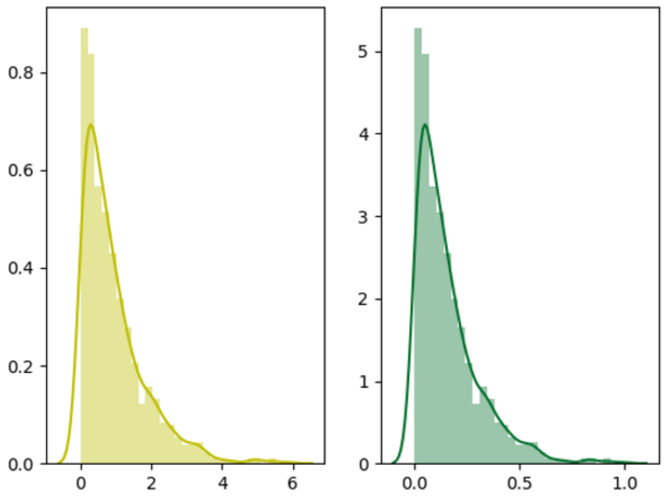


Thus, I applied SMOTE resample on the features and labels, it will resample samples for each label to remove the class imbalance problem.



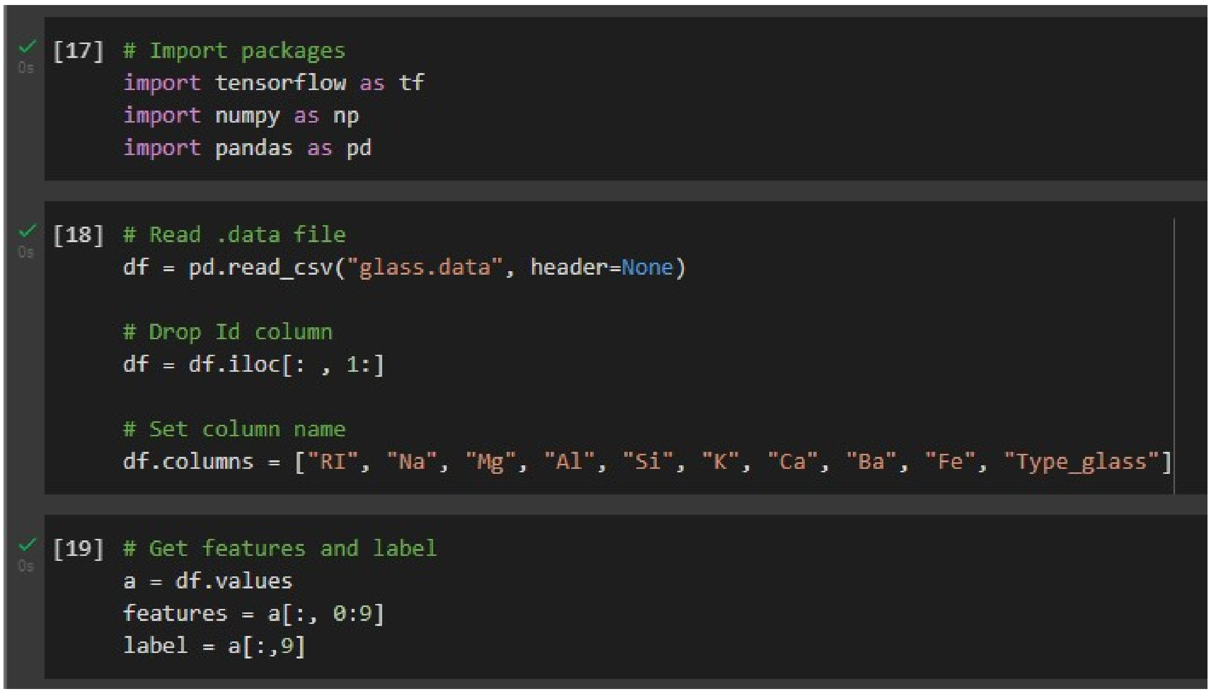
```
[24] # Perform data normalization
     from sklearn import preprocessing

     features_nz = preprocessing.normalize(features_)
     print("Normalized Data = ", features_nz)

     Normalized Data =  [[2.06246130e-02 1.84955866e-01 6.08835658e-02 ... 1.18648374e-01
       0.00000000e+00 0.00000000e+00]
      [2.03504715e-02 1.86258689e-01 4.82743903e-02 ... 1.04996799e-01
       0.00000000e+00 0.00000000e+00]
      [2.02815444e-02 1.80987281e-01 4.74874240e-02 ... 1.04071031e-01
       0.00000000e+00 0.00000000e+00]
      ...
      [2.02697647e-02 1.89054044e-01 0.00000000e+00 ... 1.22389580e-01
       1.27467837e-02 1.58398278e-04]
      [2.00670285e-02 1.95691202e-01 0.00000000e+00 ... 1.14287267e-01
       7.99003223e-03 0.00000000e+00]
      [2.04768863e-02 1.81648361e-01 4.45920764e-02 ... 1.09648623e-01
       3.23630362e-03 0.00000000e+00]]
```

After that, I used Scikit-learn package's command to apply normalization to the balanced features. Machine learning uses data normalization to reduce the sensitivity of model training to feature scale. This enables our model to converge to better weights, which results in a model that is more accurate.



Example, Left: Original data, Right: Normalized data

Printed the source code and a short explanation of each part

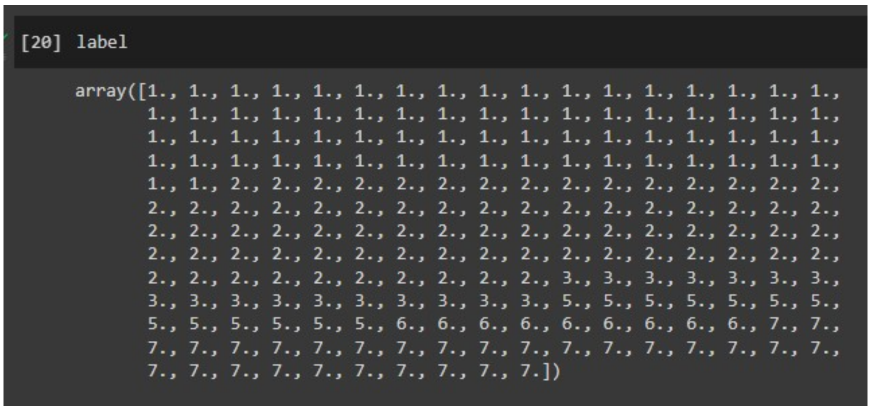


```
[17] # Import packages
     import tensorflow as tf
     import numpy as np
     import pandas as pd

[18] # Read .data file
     df = pd.read_csv("glass.data", header=None)

     # Drop Id column
     df = df.iloc[: , 1:]

     # Set column name
     df.columns = ["RI", "Na", "Mg", "Al", "Si", "K", "Ca", "Ba", "Fe", "Type_glass"]

[19] # Get features and label
     a = df.values
     features = a[:, 0:9]
     label = a[:,9]
```

First, I imported the main packages, including TensorFlow, to work on neural networks. NumPy, to work on arrays, and Pandas to work on Data Frames.

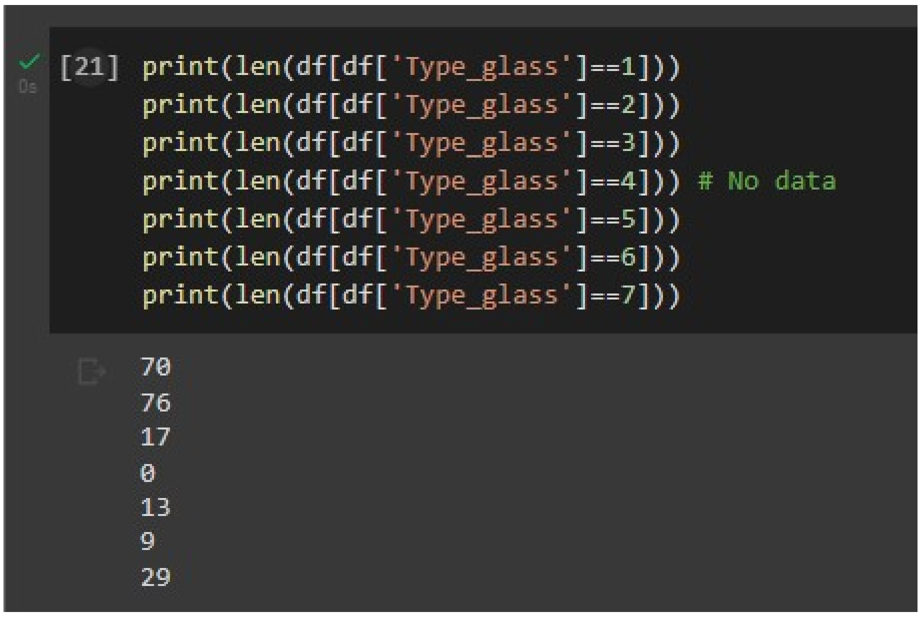
Then, I read the .data file, that is already in the CSV format, dropped the Id column because it is not necessary for the neural network training. Then give column name for each column.

After that, I apply code to get features and label, first 9 columns are the features (0:9 = column 0-8), and the last column is label (type of glass).



```
[20] label

     array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
            1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
            1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
            1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
            1., 1., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
            2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
            2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
            2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
            2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3.,
            3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 5., 5., 5., 5., 5., 5., 5.,
            5., 5., 5., 5., 5., 5., 6., 6., 6., 6., 6., 6., 6., 6., 6., 7., 7.,
            7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7.,
            7., 7., 7., 7., 7., 7., 7., 7., 7., 7.])
```

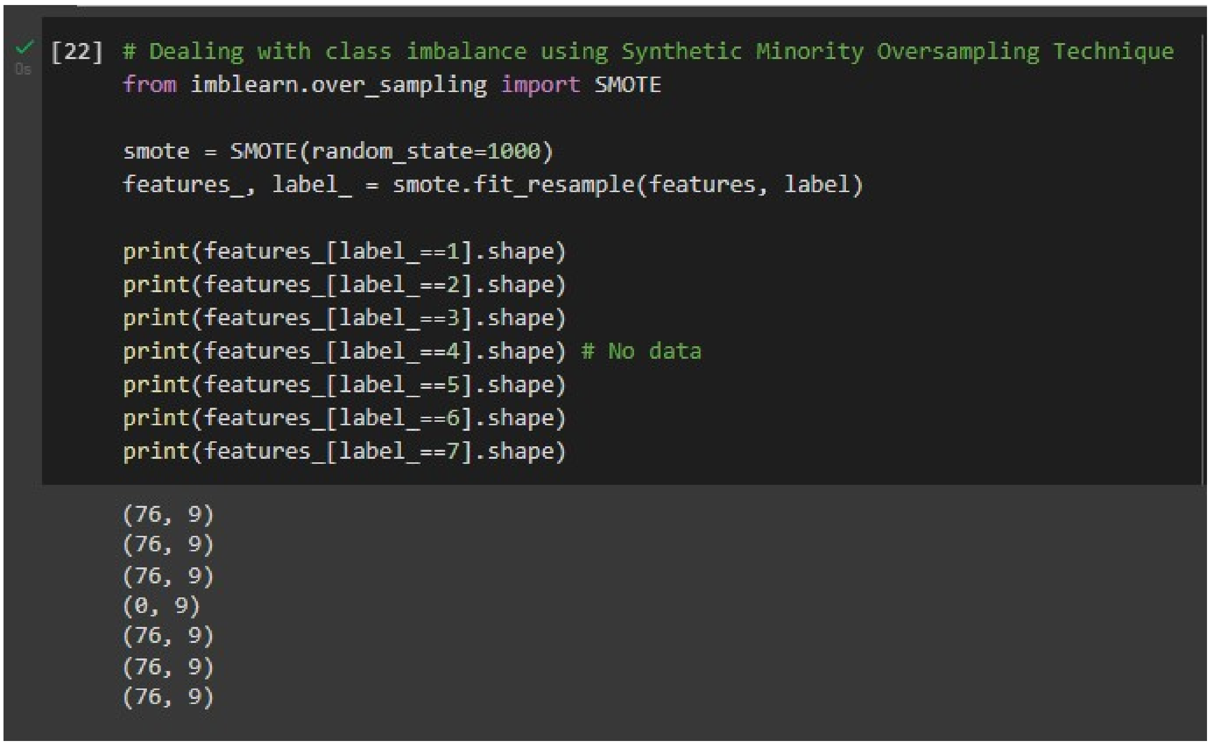
There is no sample for label 4, as the instruction said.



```
[21] print(len(df[df['Type_glass']==1]))
     print(len(df[df['Type_glass']==2]))
     print(len(df[df['Type_glass']==3]))
     print(len(df[df['Type_glass']==4])) # No data
     print(len(df[df['Type_glass']==5]))
     print(len(df[df['Type_glass']==6]))
     print(len(df[df['Type_glass']==7]))
```

```
70
76
17
0
13
9
29
```

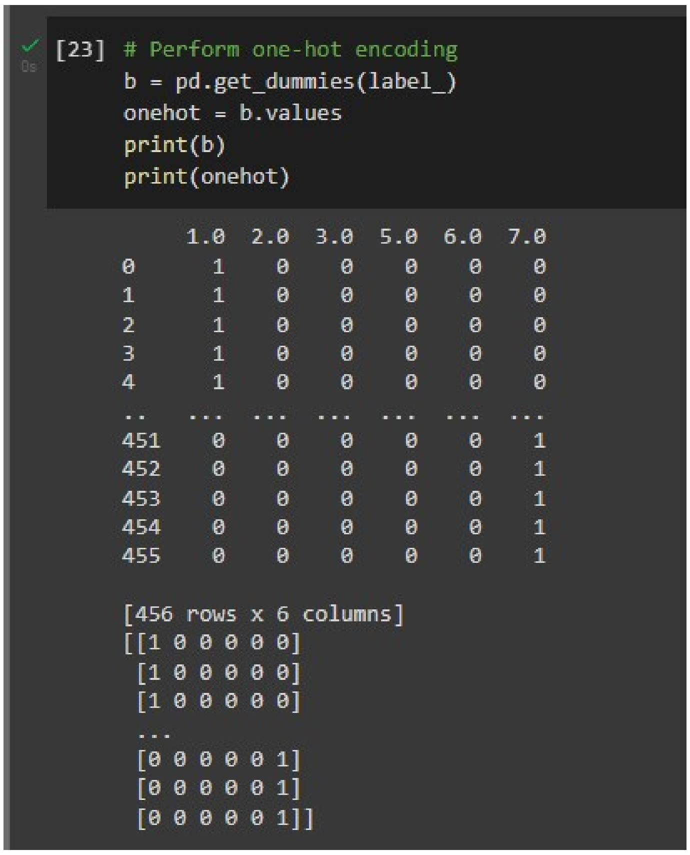
I check how many samples for each label by printing out len of df according to the label number. You can see that class balancing need to be performed.



```
[22] # Dealing with class imbalance using Synthetic Minority Oversampling Technique
     from imblearn.over_sampling import SMOTE

     smote = SMOTE(random_state=1000)
     features_, label_ = smote.fit_resample(features, label)

     print(features_[label_==1].shape)
     print(features_[label_==2].shape)
     print(features_[label_==3].shape)
     print(features_[label_==4].shape) # No data
     print(features_[label_==5].shape)
     print(features_[label_==6].shape)
     print(features_[label_==7].shape)
```

```
(76, 9)
(76, 9)
(76, 9)
(0, 9)
(76, 9)
(76, 9)
(76, 9)
```

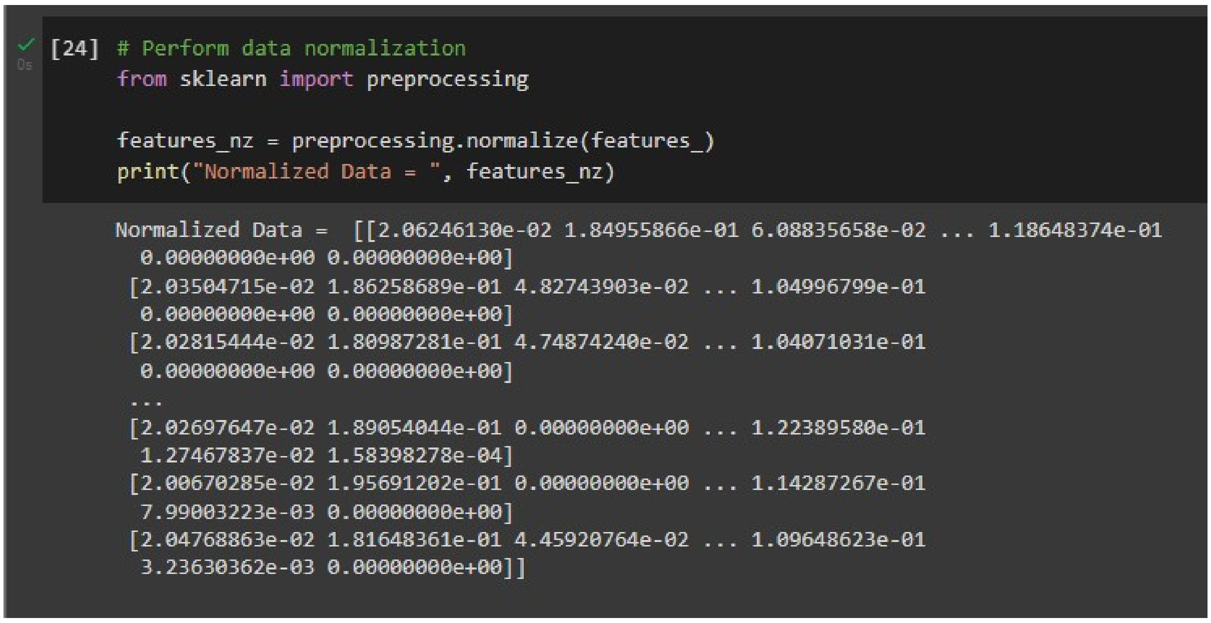
After that, I performed class balancing using SMOTE techniques, as mentioned earlier.



```
[23] # Perform one-hot encoding
     b = pd.get_dummies(label_)
     onehot = b.values
     print(b)
     print(onehot)
```

```
     1.0  2.0  3.0  5.0  6.0  7.0
0      1    0    0    0    0    0
1      1    0    0    0    0    0
2      1    0    0    0    0    0
3      1    0    0    0    0    0
4      1    0    0    0    0    0
..   ...  ...  ...  ...  ...  ...
451    0    0    0    0    0    1
452    0    0    0    0    0    1
453    0    0    0    0    0    1
454    0    0    0    0    0    1
455    0    0    0    0    0    1

[456 rows x 6 columns]
[[1 0 0 0 0 0]
 [1 0 0 0 0 0]
 [1 0 0 0 0 0]
 ...
 [0 0 0 0 0 1]
 [0 0 0 0 0 1]
 [0 0 0 0 0 1]]
```

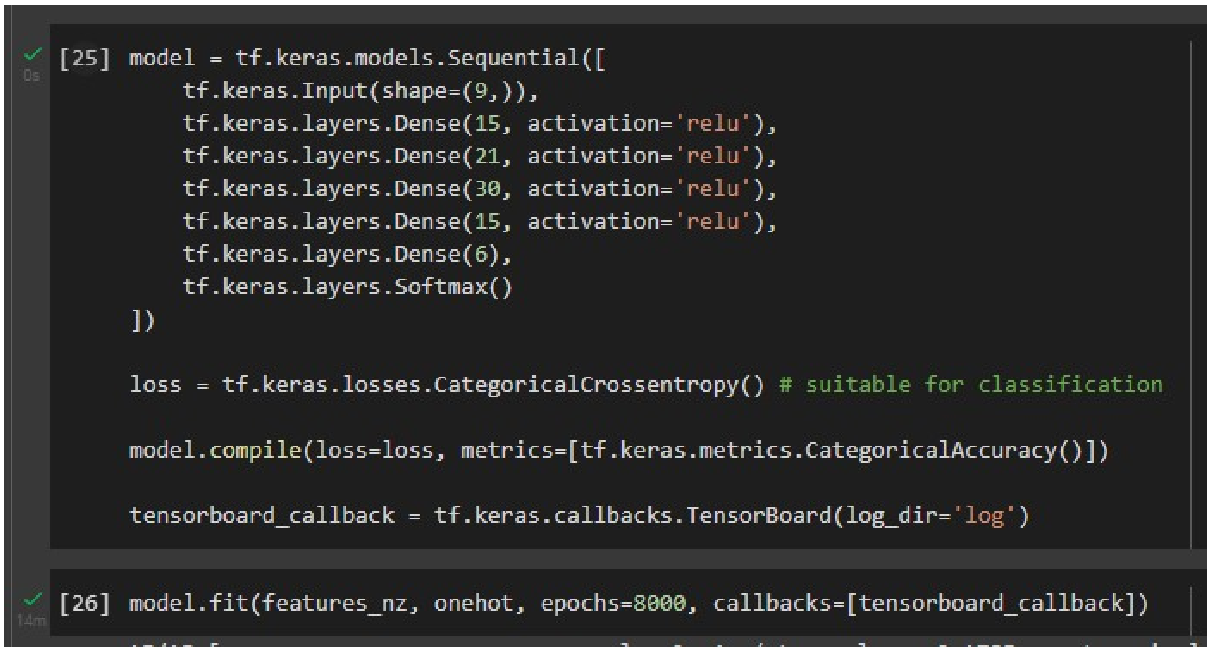
After we get the balanced features and labels, apply Pandas' command to perform one-hot encoding on new label.



```
[24] # Perform data normalization
     from sklearn import preprocessing

     features_nz = preprocessing.normalize(features_)
     print("Normalized Data = ", features_nz)
```

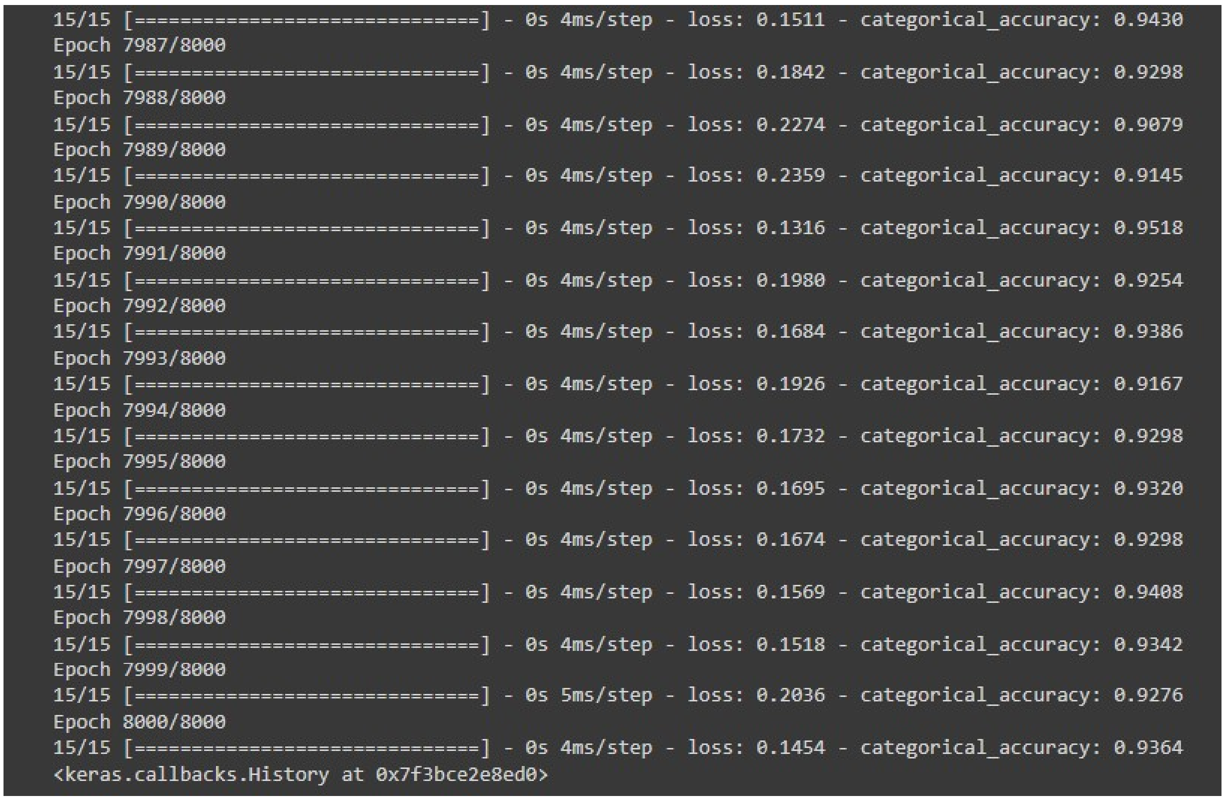
```
Normalized Data =  [[2.06246130e-02 1.84955866e-01 6.08835658e-02 ... 1.18648374e-01
  0.00000000e+00 0.00000000e+00]
 [2.03504715e-02 1.86258689e-01 4.82743903e-02 ... 1.04996799e-01
  0.00000000e+00 0.00000000e+00]
 [2.02815444e-02 1.80987281e-01 4.74874240e-02 ... 1.04071031e-01
  0.00000000e+00 0.00000000e+00]
 ...
 [2.02697647e-02 1.89054044e-01 0.00000000e+00 ... 1.22389580e-01
  1.27467837e-02 1.58398278e-04]
 [2.00670285e-02 1.95691202e-01 0.00000000e+00 ... 1.14287267e-01
  7.99003223e-03 0.00000000e+00]
 [2.04768863e-02 1.81648361e-01 4.45920764e-02 ... 1.09648623e-01
  3.23630362e-03 0.00000000e+00]]
```

After that, apply normalization using scikit-learn command as mentioned earlier.



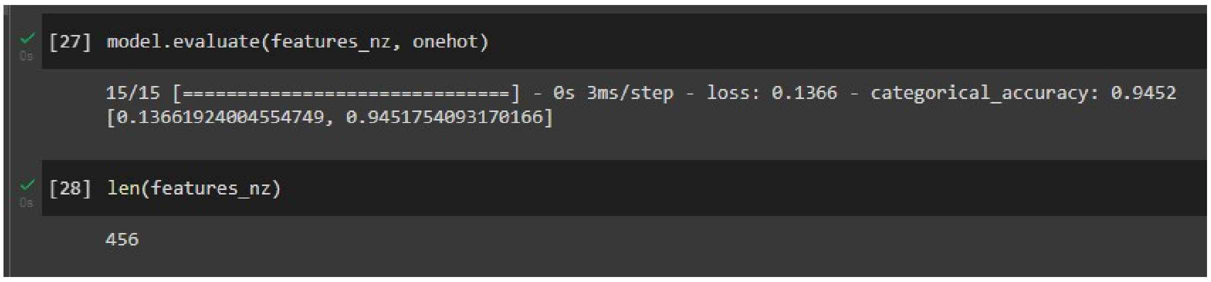
```
[25] model = tf.keras.models.Sequential([
         tf.keras.Input(shape=(9,)),
         tf.keras.layers.Dense(15, activation='relu'),
         tf.keras.layers.Dense(21, activation='relu'),
         tf.keras.layers.Dense(30, activation='relu'),
         tf.keras.layers.Dense(15, activation='relu'),
         tf.keras.layers.Dense(6),
         tf.keras.layers.Softmax()
     ])

     loss = tf.keras.losses.CategoricalCrossentropy() # suitable for classification

     model.compile(loss=loss, metrics=[tf.keras.metrics.CategoricalAccuracy()])

     tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir='log')

[26] model.fit(features_nz, onehot, epochs=8000, callbacks=[tensorboard_callback])
```

Now it's time to build neural networks. I specified the hidden layers as shown. There are 9 features, so the input shape is 9. I use ReLU activation function for every hidden layer, except for the last one. Then apply the Softmax function. The loss function is Categorical Cross Entropy, it is the one suitable for classification problem. For accuracy metric, I use Categorical Accuracy, it calculates how often predictions match one-hot labels. Then, fit the model with 8000 epochs.



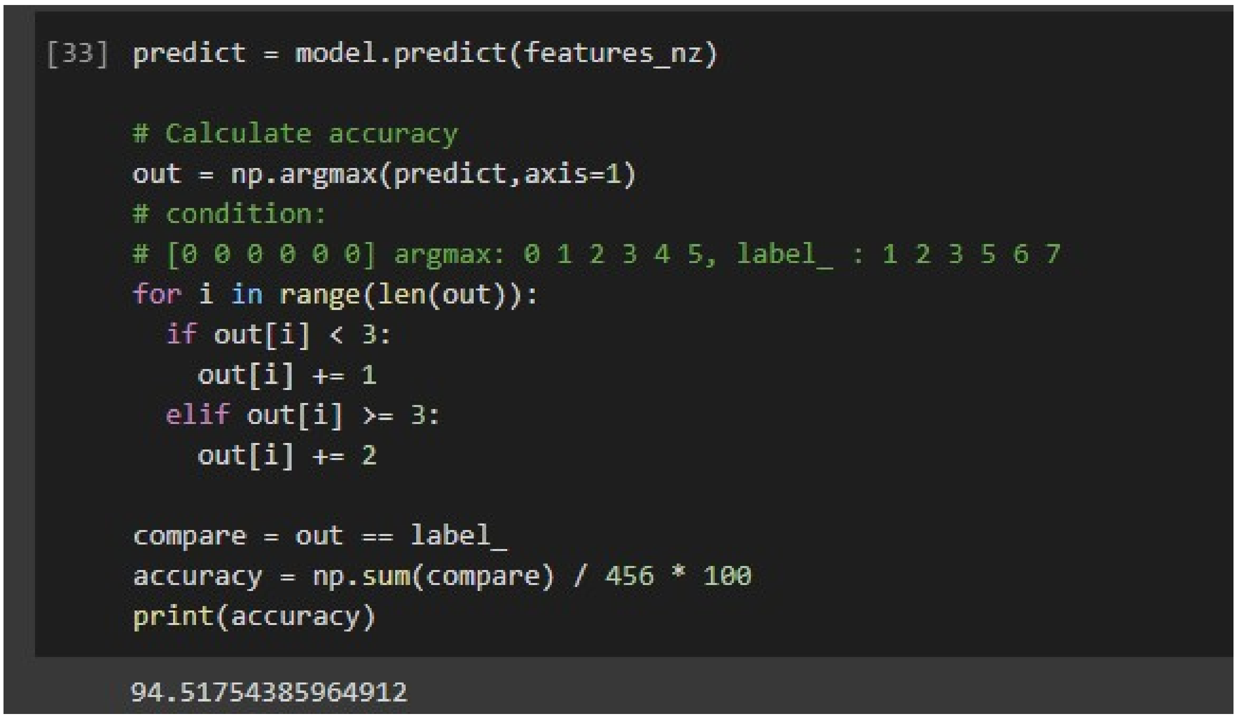
```
15/15 [==============================] - 0s 4ms/step - loss: 0.1511 - categorical_accuracy: 0.9430
Epoch 7987/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1842 - categorical_accuracy: 0.9298
Epoch 7988/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.2274 - categorical_accuracy: 0.9079
Epoch 7989/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.2359 - categorical_accuracy: 0.9145
Epoch 7990/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1316 - categorical_accuracy: 0.9518
Epoch 7991/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1980 - categorical_accuracy: 0.9254
Epoch 7992/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1684 - categorical_accuracy: 0.9386
Epoch 7993/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1926 - categorical_accuracy: 0.9167
Epoch 7994/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1732 - categorical_accuracy: 0.9298
Epoch 7995/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1695 - categorical_accuracy: 0.9320
Epoch 7996/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1674 - categorical_accuracy: 0.9298
Epoch 7997/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1569 - categorical_accuracy: 0.9408
Epoch 7998/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1518 - categorical_accuracy: 0.9342
Epoch 7999/8000
15/15 [==============================] - 0s 5ms/step - loss: 0.2036 - categorical_accuracy: 0.9276
Epoch 8000/8000
15/15 [==============================] - 0s 4ms/step - loss: 0.1454 - categorical_accuracy: 0.9364
<keras.callbacks.History at 0x7f3bce2e8ed0>
```

Wait until it finishes training procedure.



```
[27] model.evaluate(features_nz, onehot)

     15/15 [==============================] - 0s 3ms/step - loss: 0.1366 - categorical_accuracy: 0.9452
     [0.13661924004554749, 0.9451754093170166]

[28] len(features_nz)

     456
```

Evaluate model loss and accuracy. Also, I use command len(features_nz) to use for accuracy calculation further.

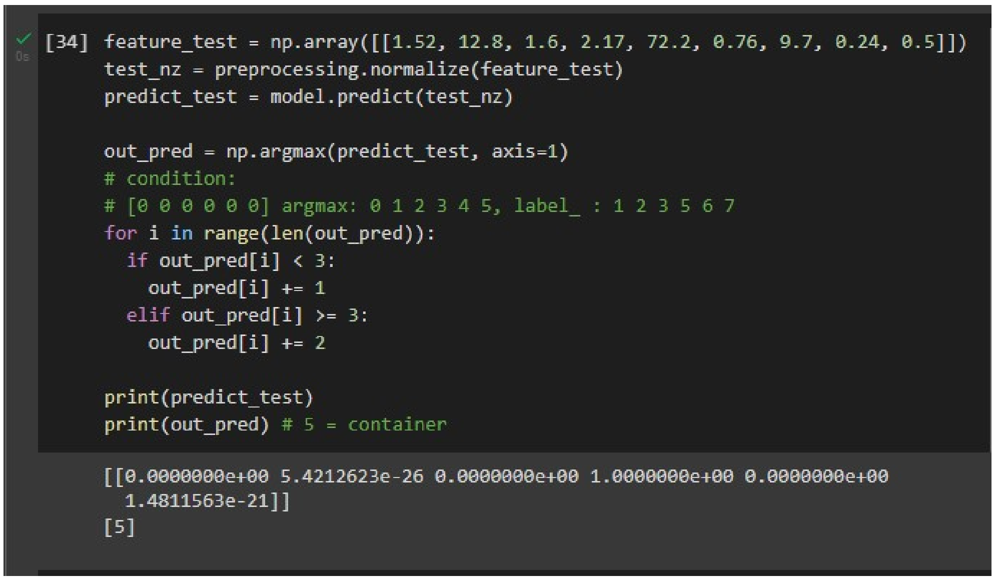


```
[33] predict = model.predict(features_nz)

     # Calculate accuracy
     out = np.argmax(predict,axis=1)
     # condition:
     # [0 0 0 0 0 0] argmax: 0 1 2 3 4 5, label_ : 1 2 3 5 6 7
     for i in range(len(out)):
       if out[i] < 3:
         out[i] += 1
       elif out[i] >= 3:
         out[i] += 2

     compare = out == label_
     accuracy = np.sum(compare) / 456 * 100
     print(accuracy)

     94.51754385964912
```

Now, try to predict on training set. The problems are

1. Label did not start with 0
2. There was no ‘4’ in the label

But the ‘out’ still treats as [0 0 0 0 0 1] for example. So, I manually rearrange the values output. For instance, if the ‘out’ output is ‘5’, it means that, the prediction is something like [0 0.01 0 0 0 0.99], the highest probability is 0.99, so if applied np.argmax command, it will return ‘5’ (since its 0-5). But the ‘5’ returned doesn’t mean it is the label ‘5’. In fact, it is labeled ‘7’. Thus, if out is less than 3 (0, 1, 2), I add 1 to it (becomes 1 2 3). But if its more than or equal to 3 (3, 4, 5), I add 2 to it (becomes 5, 6, 7). After that, we compare the true/false with the training data, sum the true (1), divide it with the number of dataset (456) and multiply by 100, we get the accuracy number (Nearly the same as using Keras’ metrics).

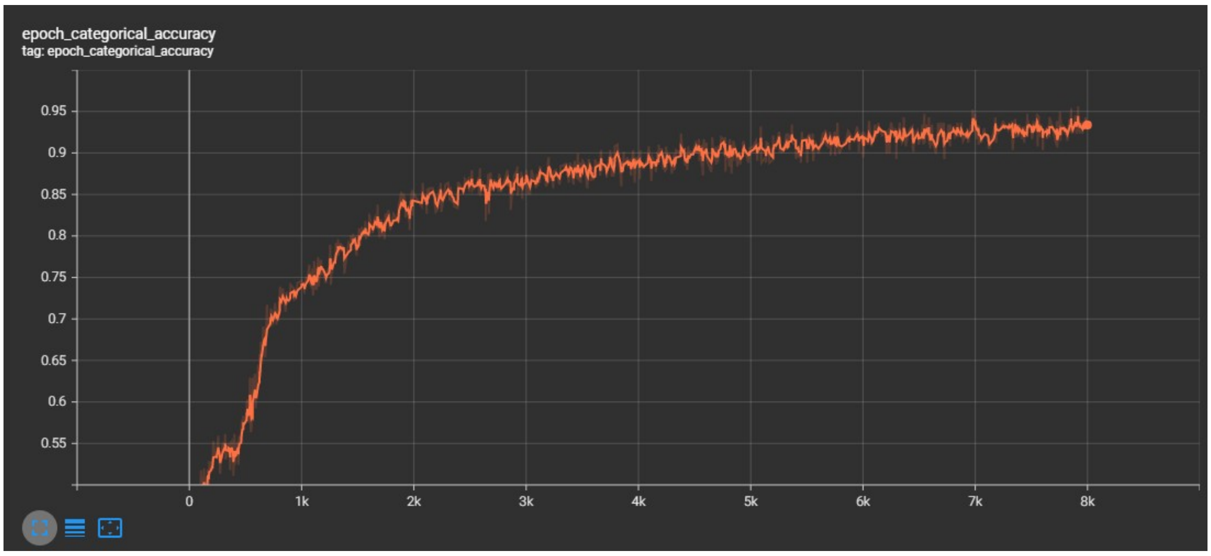


```
[34] feature_test = np.array([[1.52, 12.8, 1.6, 2.17, 72.2, 0.76, 9.7, 0.24, 0.5]])
     test_nz = preprocessing.normalize(feature_test)
     predict_test = model.predict(test_nz)

     out_pred = np.argmax(predict_test, axis=1)
     # condition:
     # [0 0 0 0 0 0] argmax: 0 1 2 3 4 5, label_ : 1 2 3 5 6 7
     for i in range(len(out_pred)):
       if out_pred[i] < 3:
         out_pred[i] += 1
       elif out_pred[i] >= 3:
         out_pred[i] += 2

     print(predict_test)
     print(out_pred) # 5 = container
```

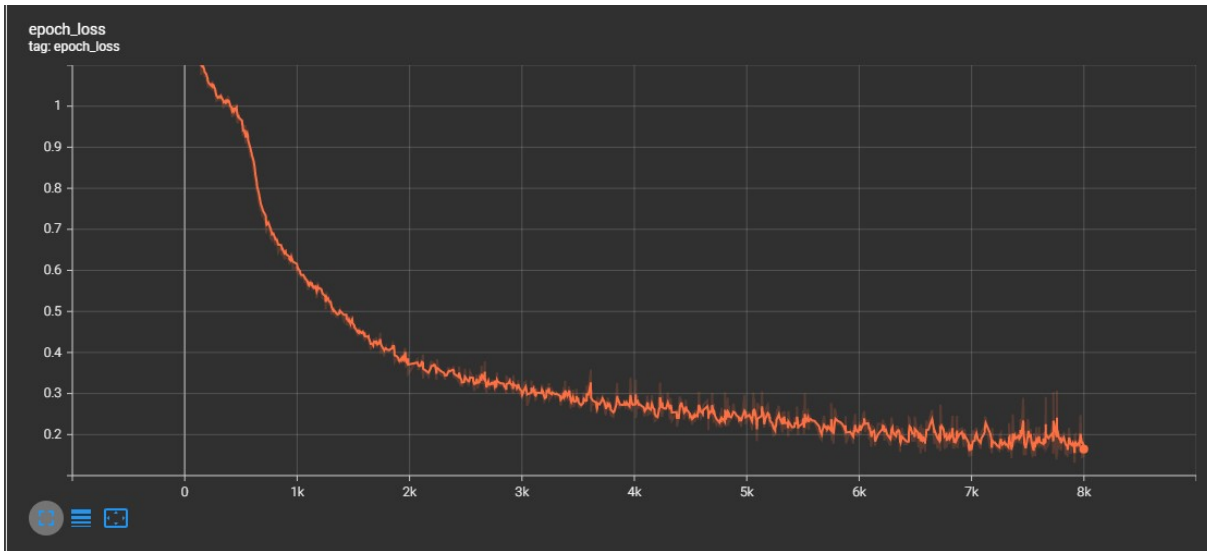
```
[[0.0000000e+00 5.4212623e-26 0.0000000e+00 1.0000000e+00 0.0000000e+00
  1.4811563e-21]]
[5]
```

Then, it's time to test the given features (1.52, 12.8, 1.6, 2.17, 72.2, 0.76, 9.7, 0.24, 0.5). Set it as NumPy array. But we have to normalize it first, since our training dataset is also normalized. Then, do the same procedure and print out the 'out_pred' to know which one has the highest probability (argmax). It seems that it shows '5' (5 = container). If we print out predict_test, it will show the softmax function probability. You can see that label '5' has the highest probability.

Graph Results (2 graphs)

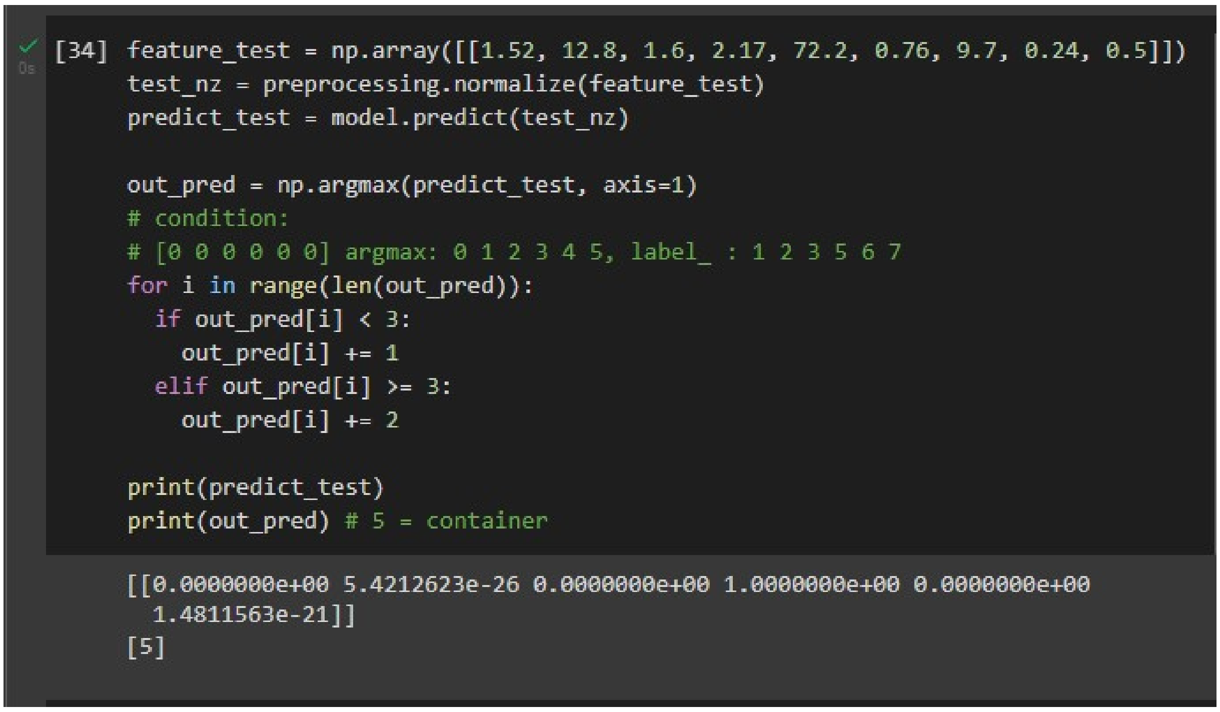


The training data categorical accuracy graph (8000 epoch)



The training data loss graph (8000 epoch)

Answer which types of glass is the sample and show the probability value



```
[34] feature_test = np.array([[1.52, 12.8, 1.6, 2.17, 72.2, 0.76, 9.7, 0.24, 0.5]])
     test_nz = preprocessing.normalize(feature_test)
     predict_test = model.predict(test_nz)

     out_pred = np.argmax(predict_test, axis=1)
     # condition:
     # [0 0 0 0 0 0] argmax: 0 1 2 3 4 5, label_ : 1 2 3 5 6 7
     for i in range(len(out_pred)):
       if out_pred[i] < 3:
         out_pred[i] += 1
       elif out_pred[i] >= 3:
         out_pred[i] += 2

     print(predict_test)
     print(out_pred) # 5 = container
```

```
[[0.0000000e+00 5.4212623e-26 0.0000000e+00 1.0000000e+00 0.0000000e+00
  1.4811563e-21]]
[5]
```

It seems that it shows ‘5’ (5 = containers). If we print out predict_test, it will show the softmax function probability. You can see that label ‘5’ has the highest probability. The probability value is nearly 1.

-- 1 building_windows_float_processed -- 2 building_windows_non_float_processed

-- 3 vehicle_windows_float_processed

-- 4 vehicle_windows_non_float_processed (none in this database)

-- 5 containers

-- 6 tableware -- 7 headlamps