MATH 6373 Spring 2021 MSDS Homework 2

Jacob Gogan
Sara Nafaryeh
Thomas Su

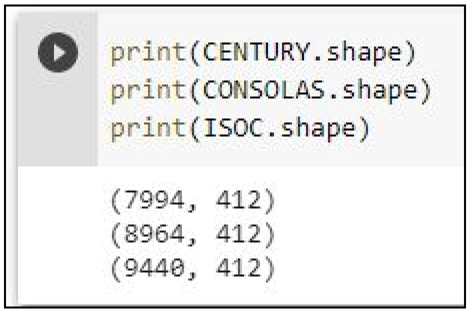
All authors played an equal part

Introduction

In this report, we have selected three font data sets consisting of digitized images of typed characters and *using MLP to predict one of the 3 font classes*. As discussed in HW1, Multilayer Perceptron (MLP) is a type of feedforward Artificial Neural Network Model that introduces one or more hidden layers on the basis of a single-layer neural network. The hidden layer is located between the input layer and the output layer. Therefore, the neurons in the hidden layer are fully connected to each input in the input layer, and the neurons in the output layer and each neuron in the hidden layer are also fully connected. Also, the input layer doesn't involve any calculation.

The files were downloaded from a zip file from the following link: https://archive.ics.uci.edu/ml/machine-learning-databases/00417/

The font types that we chose were: CENTURY, CONSOLAS, and ISOC. Each has 412 column features along with total observed cases of 7994, 8964, and 9440 respectively.



```
print(CENTURY.shape)
print(CONSOLAS.shape)
print(ISOC.shape)

(7994, 412)
(8964, 412)
(9440, 412)
```

Each case describes numerically a digitized image of some specific character typed in each of the fonts. The images have 20x20 = 400 pixel sizes, each with its own “gray level” indicated by an integer value of 0 to 255. All the fonts have 412 feature columns, where 400 of them describe the 400 pixels named:

{ r0c0, r0c1,r0c2, … , r19,c17, r19c18, r19c19}

"rLcM" = gray level image intensity for pixel in position **{Row L, Column M}**.

Finally, the last important columns in the data set we will be looking at are the strength and italic columns. The strength column lists 0.4 = normal character and 0.7 = bold character. The italic column lists 1 = italic character and 0 = normal character.

STEP 0: Data Setup

The following steps are taken in python to get our data ready for the steps that follow. Other than the 400 columns that are associated with the pixels, the data set font files each have the following 12 names:

{ font, fontVariant, m_label, strength, italic, orientation, m_top, m_left, originalH, originalW, h, w }

Of these 12 we need to discard the following 9:

{fontVariant, m_label, orientation, m_top, m_left, originalH, originalW, h, w}

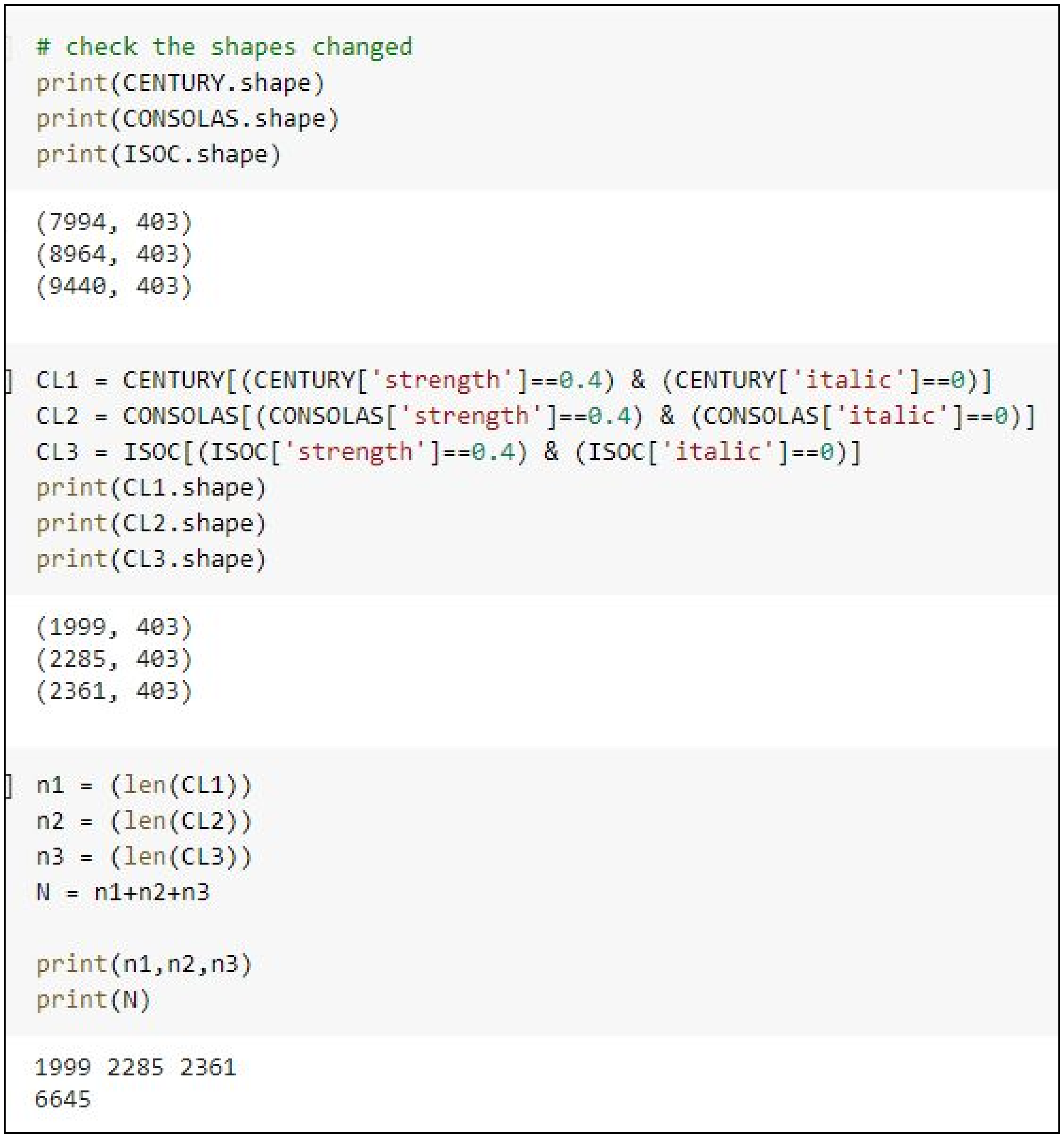
And keep the following 3: ***{font, strength, italic}*** as well as the 400 pixel columns named: ***{ r0c0, r0c1,r0c2, … , r19,c17, r19c18, r19c19}*** therefore we are left with 403 columns. After these steps are completed, we define three CLASSES on images of the "normal" character where we extract all the rows in which our three fonts have both strength of 0.4 and italic of 0:

CL1 = all rows of **CENTURY.csv** file for which {strength = 0.4 and italic=0}

CL2 = all rows of **CONSOLAS.csv** file for which {strength = 0.4 and italic=0}

CL3 = all rows of **ISOC.csv** file for which {strength = 0.4 and italic=0}

And left with the following row outputs:



```
# check the shapes changed
print(CENTURY.shape)
print(CONSOLAS.shape)
print(ISOC.shape)
```

```
(7994, 403)
(8964, 403)
(9440, 403)
```

```
CL1 = CENTURY[(CENTURY['strength']==0.4) & (CENTURY['italic']==0)]
CL2 = CONSOLAS[(CONSOLAS['strength']==0.4) & (CONSOLAS['italic']==0)]
CL3 = ISOC[(ISOC['strength']==0.4) & (ISOC['italic']==0)]
print(CL1.shape)
print(CL2.shape)
print(CL3.shape)
```

```
(1999, 403)
(2285, 403)
(2361, 403)
```

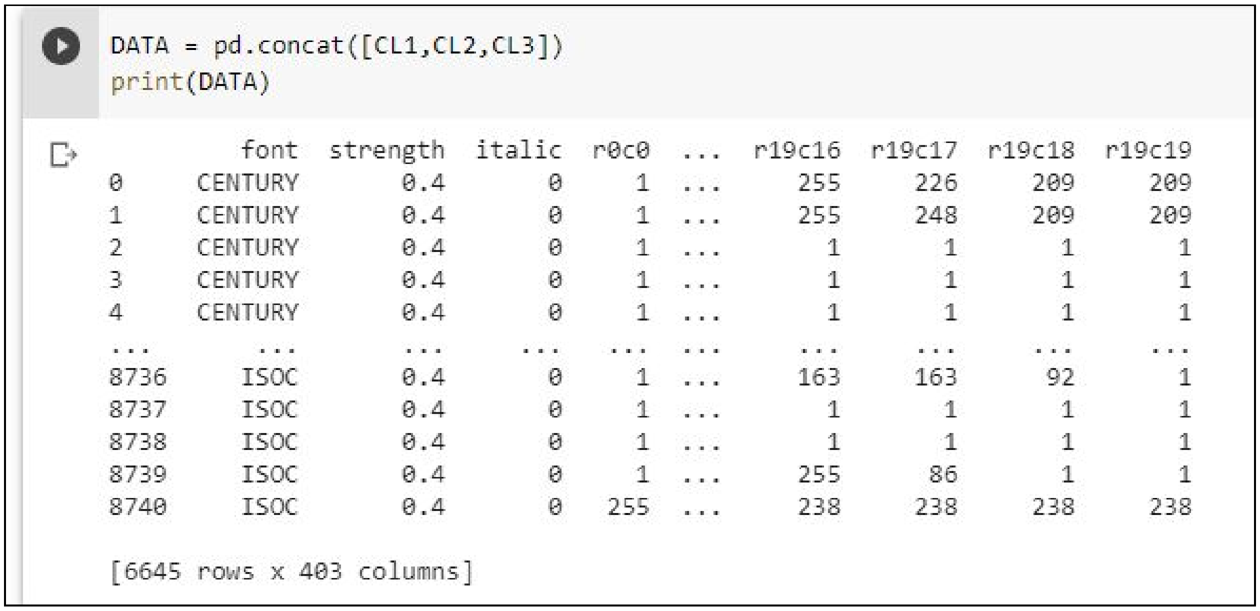
```
n1 = (len(CL1))
n2 = (len(CL2))
n3 = (len(CL3))
N = n1+n2+n3

print(n1,n2,n3)
print(N)
```

```
1999 2285 2361
6645
```

The respected row sizes for CLASS1, CLASS, and CLASS3 are named

n1, n2, n3 = 1999, 2285, 2361 and their sum → N = 6645 cases

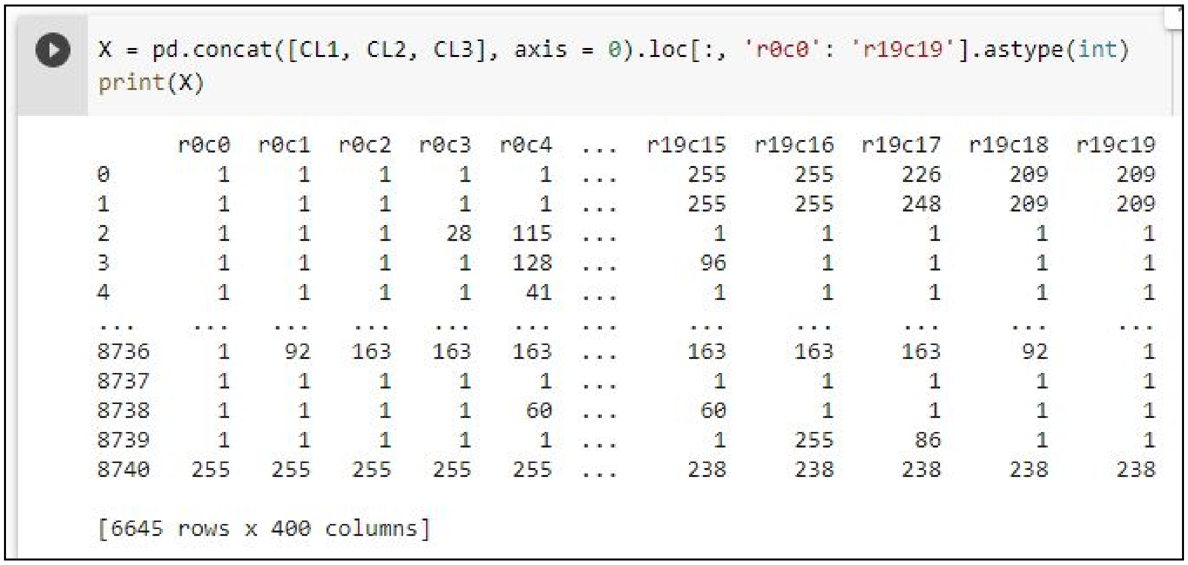
We combine them all together into a data set named DATA using the function **rbind()** and see it has dimensions of 1930 rows and 403 columns: the 403 being font, strength, italic, plus 400 pixels we will call features X1, X2, … X400. Each such feature Xj is observed N times, and its N observed values are listed in column "j" of DATA.



```
DATA = pd.concat([CL1,CL2,CL3])
print(DATA)
```

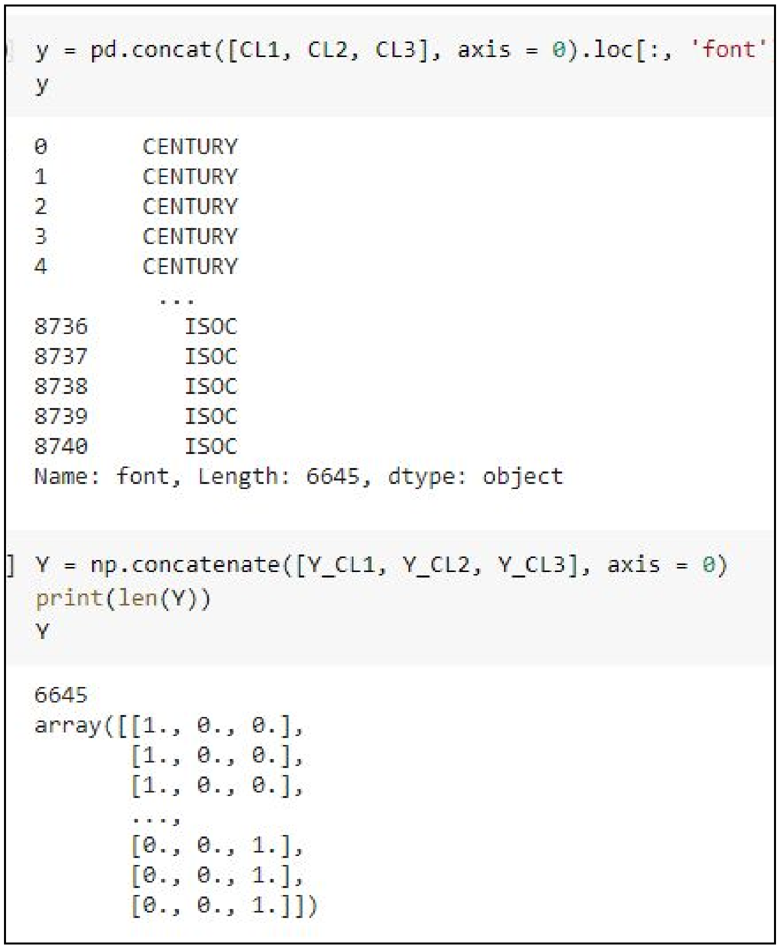
```
          font  strength  italic  r0c0  ...  r19c16  r19c17  r19c18  r19c19
0      CENTURY       0.4       0     1  ...     255     226     209     209
1      CENTURY       0.4       0     1  ...     255     248     209     209
2      CENTURY       0.4       0     1  ...       1       1       1       1
3      CENTURY       0.4       0     1  ...       1       1       1       1
4      CENTURY       0.4       0     1  ...       1       1       1       1
...        ...       ...     ...   ...  ...     ...     ...     ...     ...
8736      ISOC       0.4       0     1  ...     163     163      92       1
8737      ISOC       0.4       0     1  ...       1       1       1       1
8738      ISOC       0.4       0     1  ...       1       1       1       1
8739      ISOC       0.4       0     1  ...     255      86       1       1
8740      ISOC       0.4       0   255  ...     238     238     238     238

[6645 rows x 403 columns]
```

Setting, X = The 400 input for case n, and Y = The true output of 3 font classes as the following [1 0 0], [0 1 0], [0 0 1] one-hot encoding



```
X = pd.concat([CL1, CL2, CL3], axis = 0).loc[:, 'r0c0': 'r19c19'].astype(int)
print(X)
```

```
      r0c0  r0c1  r0c2  r0c3  r0c4  ...  r19c15  r19c16  r19c17  r19c18  r19c19
0        1     1     1     1     1  ...     255     255     226     209     209
1        1     1     1     1     1  ...     255     255     248     209     209
2        1     1     1    28   115  ...       1       1       1       1       1
3        1     1     1     1   128  ...      96       1       1       1       1
4        1     1     1     1    41  ...       1       1       1       1       1
...    ...   ...   ...   ...   ...  ...     ...     ...     ...     ...     ...
8736     1    92   163   163   163  ...     163     163     163      92       1
8737     1     1     1     1     1  ...       1       1       1       1       1
8738     1     1     1     1    60  ...      60       1       1       1       1
8739     1     1     1     1     1  ...       1     255      86       1       1
8740   255   255   255   255   255  ...     238     238     238     238     238

[6645 rows x 400 columns]
```



```
y = pd.concat([CL1, CL2, CL3], axis = 0).loc[:, 'font'
y
```

```
0       CENTURY
1       CENTURY
2       CENTURY
3       CENTURY
4       CENTURY
         ...
8736       ISOC
8737       ISOC
8738       ISOC
8739       ISOC
8740       ISOC
Name: font, Length: 6645, dtype: object
```

```
Y = np.concatenate([Y_CL1, Y_CL2, Y_CL3], axis = 0)
print(len(Y))
Y
```

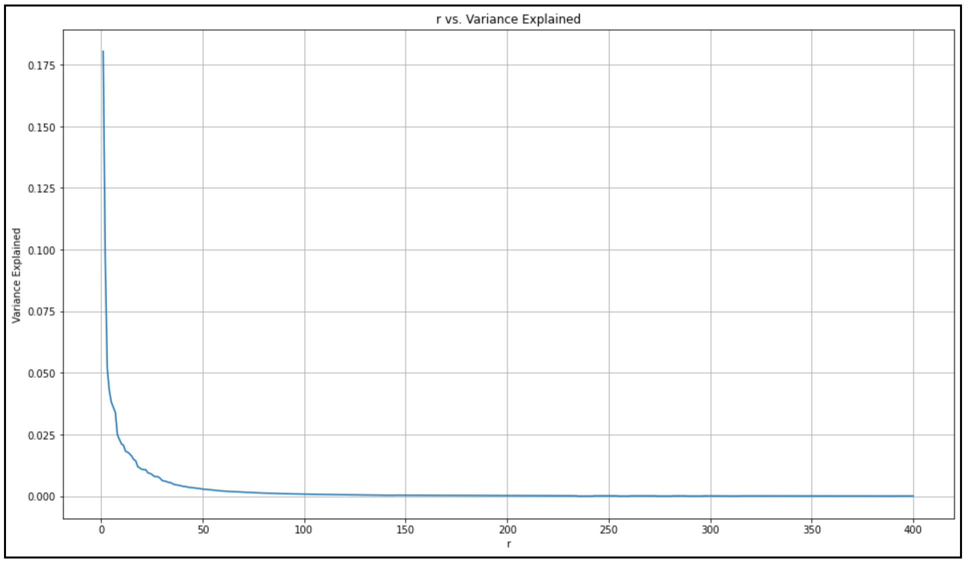
```
6645
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       ...,
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

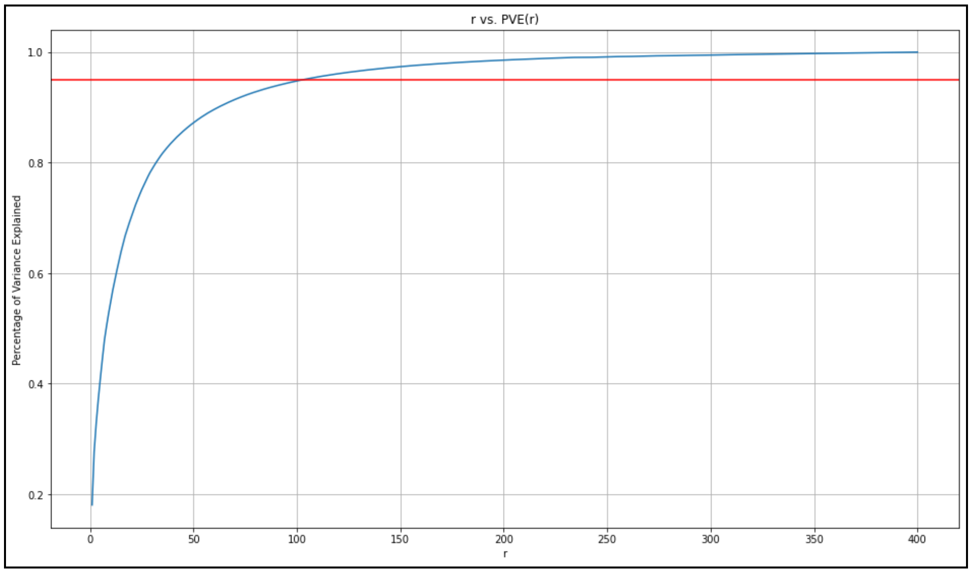
STEP 1: Lower value h0 and higher value h*

In this step, we will choose our small and large hidden values h0 and h*. From the DATA set we obtained earlier, we define a new data frame X as the training input vectors with a dim 400 features. We center and rescale all of the inputs using:

```
SX =  preprocessing.scale(X)
```

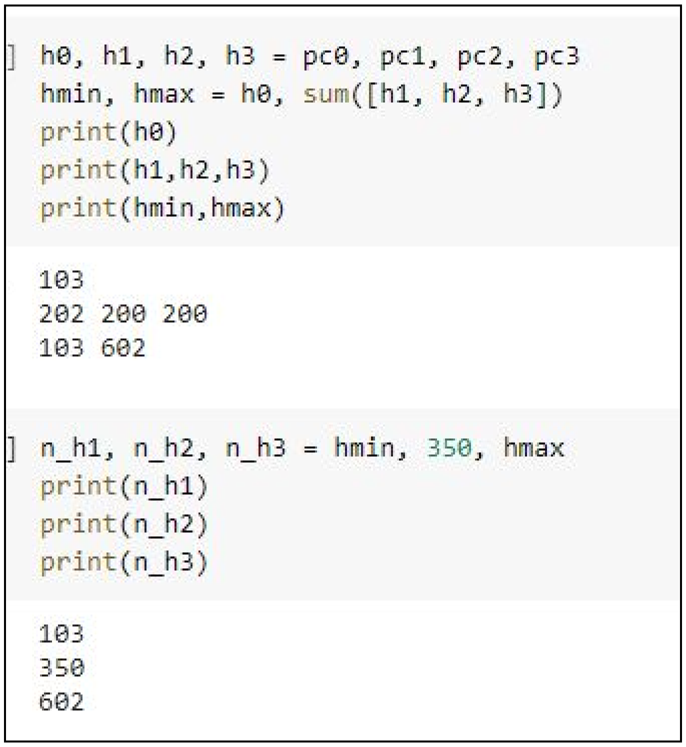
Next, we will implement Principal Component Analysis (PCA) of the true set vector Y, to calculate the small h = h0. For h0, our target percentage of variance explained (PVE) will be PVE(H0) = 95%. We find that h0 = 103. We note that h0 < k = 400 features.





To find the higher value h* of our three font classes, we will again implement PCA and calculate PVE(h*) = .99. For each class CLj, SETj = all the inputs of Xn that belong to the training inputs of CLj where j = class 1,2,3.

Using PCA we find h1 = 202, h2 = 200, and h3 = 200,such that PEV(hj) = 99% Now we have our h* = h1+h2+h3 = 602. Consequently, h* > k = 400 features. Having identifying our small h0 and large h* hidden values, we select a 3rd hidden h value between these to be 350.



```
] h0, h1, h2, h3 = pc0, pc1, pc2, pc3
  hmin, hmax = h0, sum([h1, h2, h3])
  print(h0)
  print(h1,h2,h3)
  print(hmin,hmax)
```

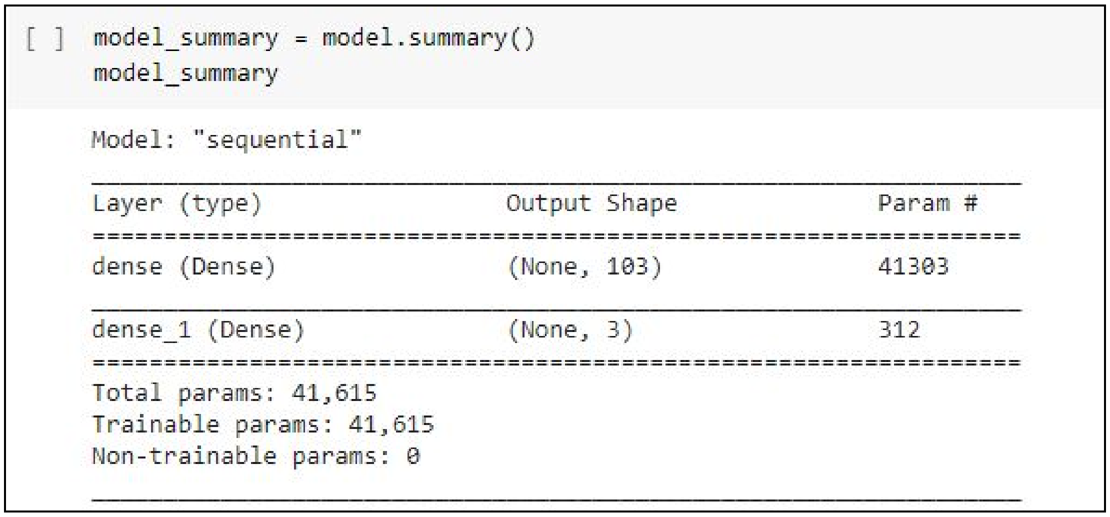
```
103
202 200 200
103 602
```

```
] n_h1, n_h2, n_h3 = hmin, 350, hmax
  print(n_h1)
  print(n_h2)
  print(n_h3)
```

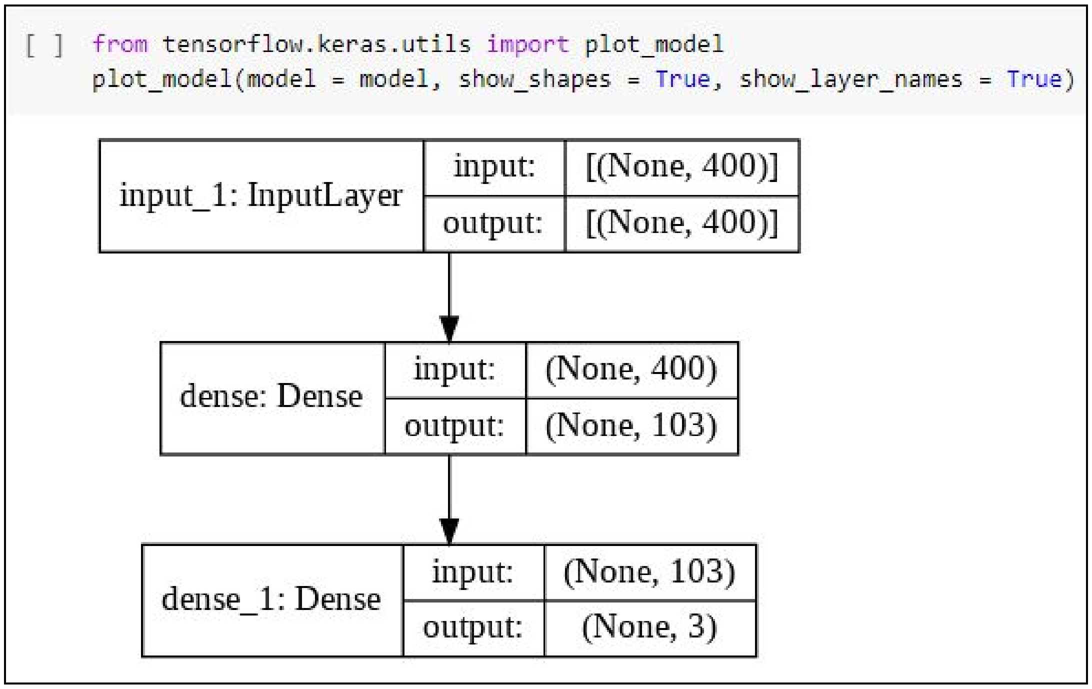
```
103
350
602
```

STEP 2: Training MLP0 and MLP*

Using the same steps taken in HW1 we train our MLP with the activation functions **ReLu** from input to hidden layer and **softmax** from hidden layer to output. We use the softmax activation function because we want to normalize our outputs and convert them from weighted sum values to probabilities that sum to one. Training MLP0 for hidden layer h0 = 103 we get the following model summary:



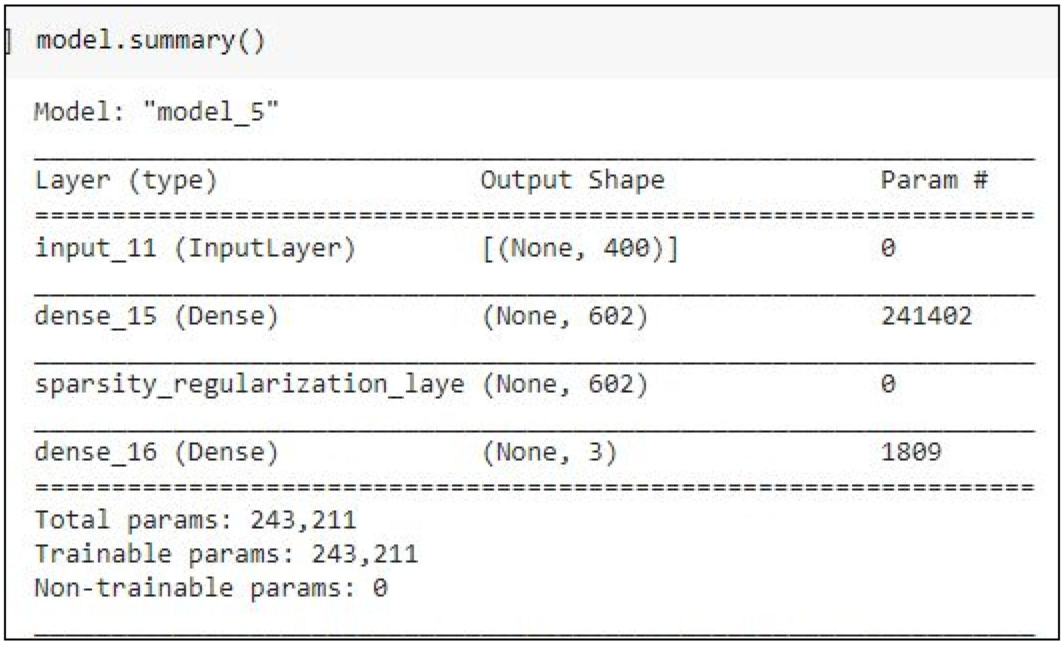
```
[ ]  model_summary = model.summary()
     model_summary

     Model: "sequential"
     _________________________________________________________________
     Layer (type)                 Output Shape              Param #
     =================================================================
     dense (Dense)                (None, 103)               41303
     _________________________________________________________________
     dense_1 (Dense)              (None, 3)                 312
     =================================================================
     Total params: 41,615
     Trainable params: 41,615
     Non-trainable params: 0
     _________________________________________________________________
```



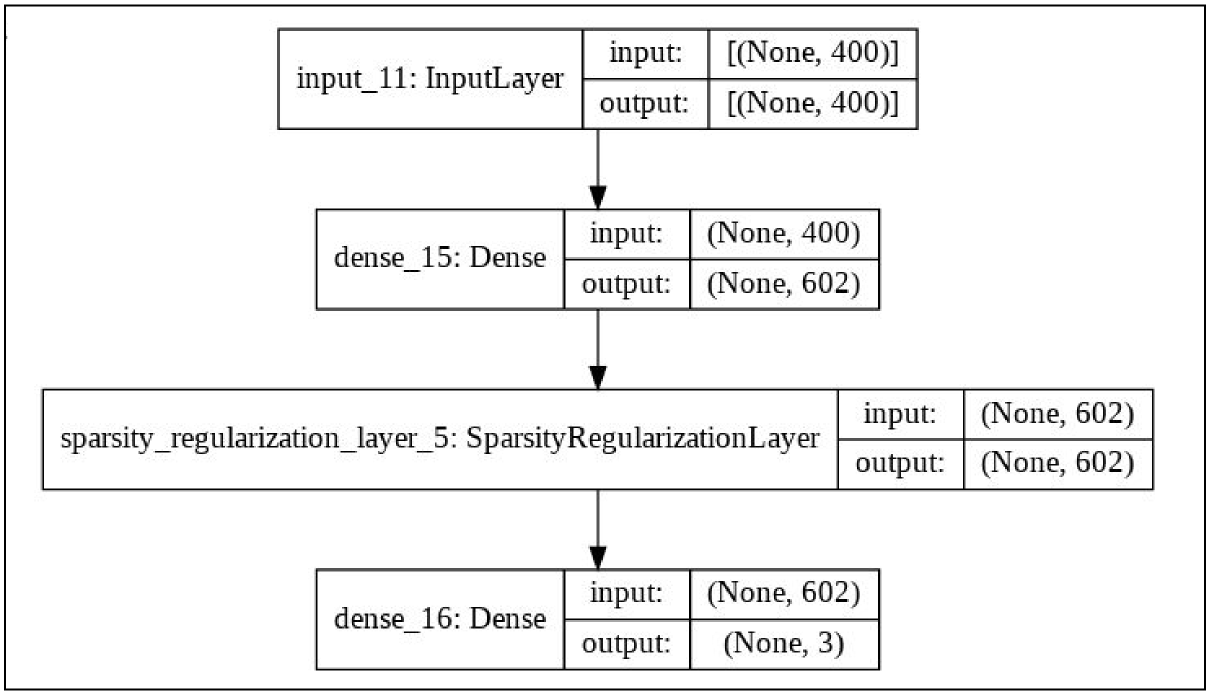
```
[ ]  from tensorflow.keras.utils import plot_model
     plot_model(model = model, show_shapes = True, show_layer_names = True)
```

In order to find MLP* for our hidden layer h* = 602, we calculate sparse autoencoders using sparsity learning. Our hidden layer h* is larger than both the input and output values. This can be an issue when some inputs can go through the model without any change to the output layer or having some of the nodes not being used at all. This can cause the model to miss some important information resulting in a misleading output. Therefore one possible solution to this is using a sparse autoencoder. This process is able to remove some of the neurons in the hidden layer, forcing the autoencoder to use all the neurons of the model. It no longer just memorizes the input through certain nodes since each run those nodes may not be the ones active. By using autoencoders for our large hidden layers, our model's output accuracy increases.

Therefore, using the TensorFlow function `SparsityRegularizationLayer()` at Tar level = 0.01, we get the following model summary and plot.



```
] model.summary()

Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_11 (InputLayer)        [(None, 400)]             0
_________________________________________________________________
dense_15 (Dense)             (None, 602)               241402
_________________________________________________________________
sparsity_regularization_laye (None, 602)               0
_________________________________________________________________
dense_16 (Dense)             (None, 3)                 1809
=================================================================
Total params: 243,211
Trainable params: 243,211
Non-trainable params: 0
_________________________________________________________________
```


input_11: InputLayer
input:
[(None, 400)]
output:
[(None, 400)]
dense_15: Dense
input:
(None, 400)
output:
(None, 602)
sparsity_regularization_layer_5: SparsityRegularizationLayer
input:
(None, 602)
output:
(None, 602)
dense_16: Dense
input:
(None, 602)
output:
(None, 3)

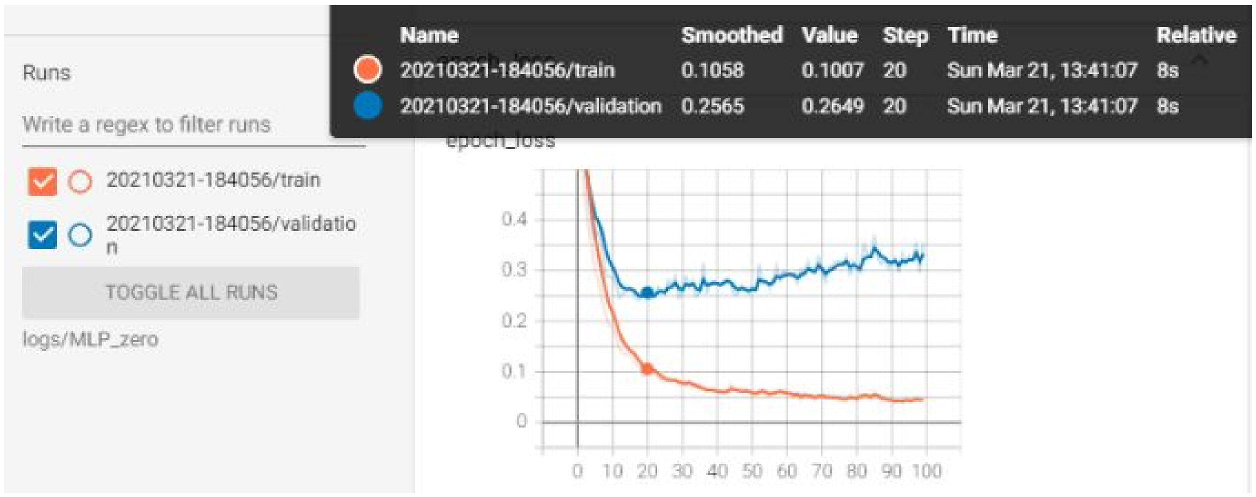
We separately monitor the training speed and quality for MLP0 and MLP* to evaluate 3 times: stabT, overT, and Tstop.

Stabilization for cross-entropy: **stabT** = The time of stabilization for cross entropy.

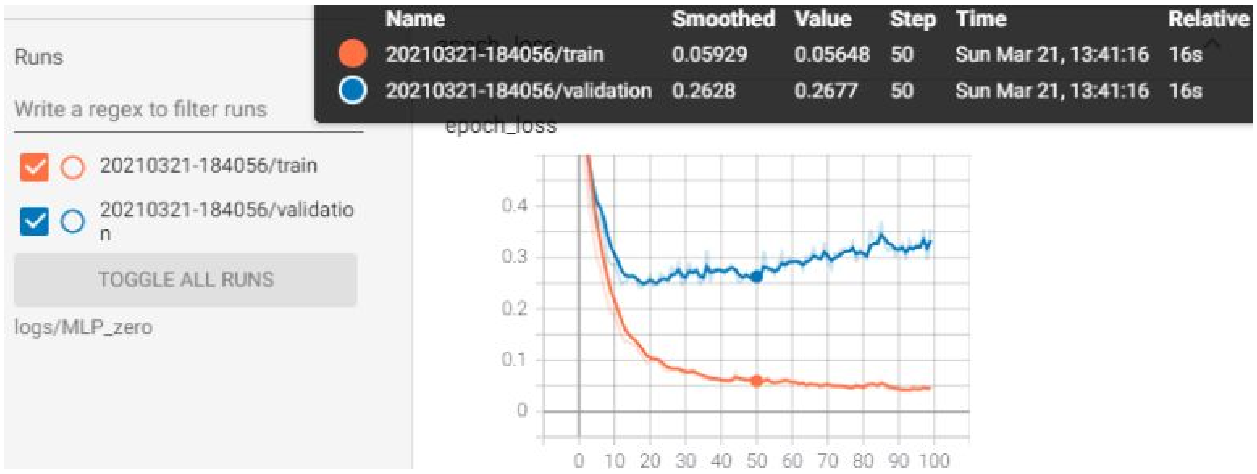
Overfit: **overfitT** = The beginning time of overfit for cross-entropy

Stopping: **Tstop** = min(overT, stabT) on the training and test set

Using pythons TensorBoard, we get the following epoch loss graph. The following are the cross-entropy values after each epoch for training and test set for MLP0. By Visual detection, we conclude stabT = 20 epochs and overT = 50 epochs, therefore Tstop = 20

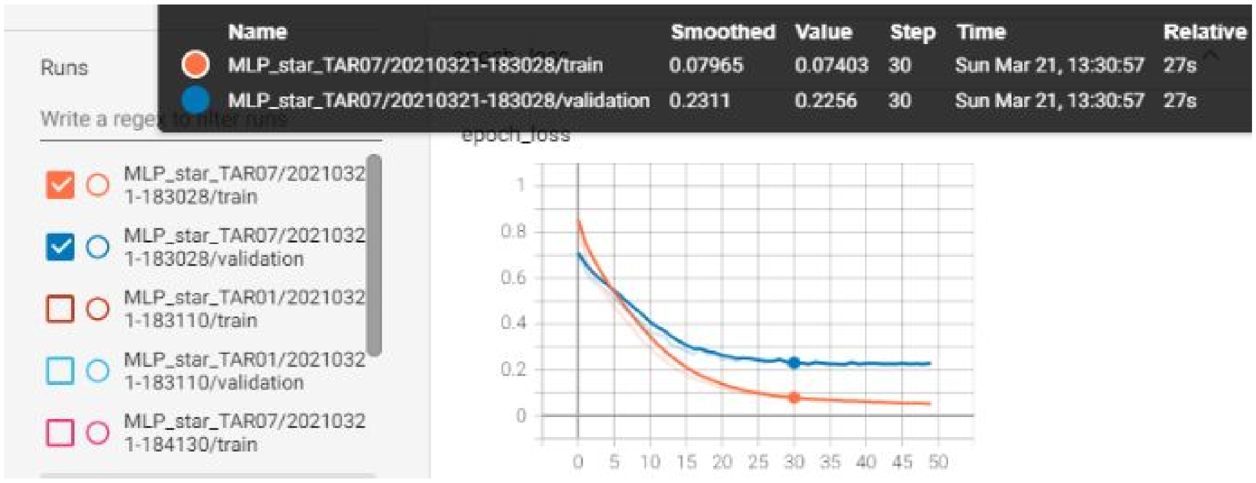


	Name	Smoothed	Value	Step	Time	Relative
●	20210321-184056/train	0.1058	0.1007	20	Sun Mar 21, 13:41:07	8s
●	20210321-184056/validation	0.2565	0.2649	20	Sun Mar 21, 13:41:07	8s



	Name	Smoothed	Value	Step	Time	Relative
●	20210321-184056/train	0.05929	0.05648	50	Sun Mar 21, 13:41:16	16s
●	20210321-184056/validation	0.2628	0.2677	50	Sun Mar 21, 13:41:16	16s

The following are the cross-entropy values after each epoch for training and test set for MLP*. The time is measured in the number of epochs. By Visual detection, we conclude stabT = 30 epochs and overT = 50 epochs, therefore Tstop = 30

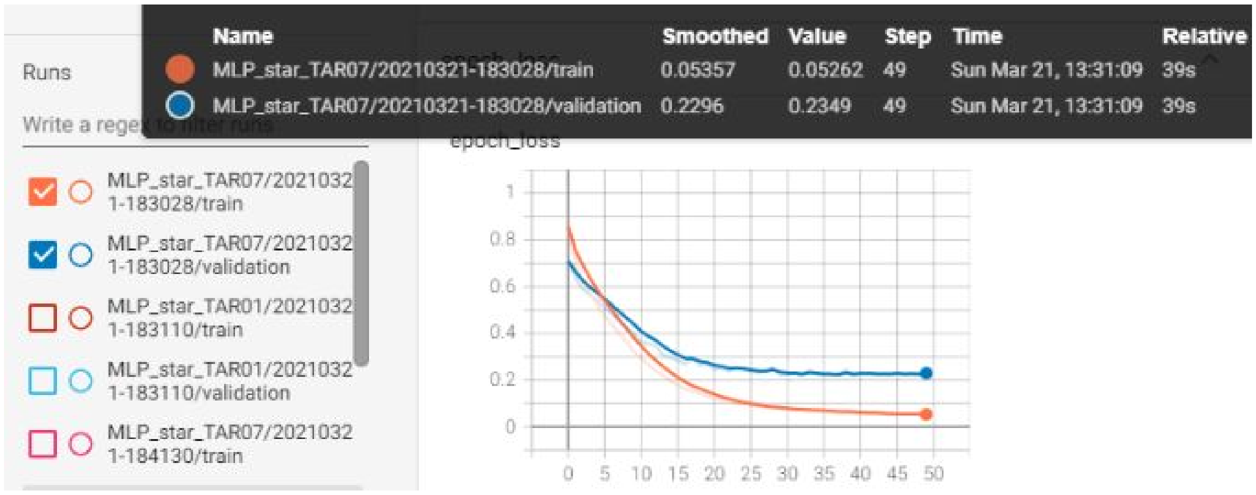


	Name	Smoothed	Value	Step	Time	Relative
●	MLP_star_TAR07/20210321-183028/train	0.07965	0.07403	30	Sun Mar 21, 13:30:57	27s
●	MLP_star_TAR07/20210321-183028/validation	0.2311	0.2256	30	Sun Mar 21, 13:30:57	27s

Runs

Write a regex to filter runs

- ✓ MLP_star_TAR07/20210321-183028/train
- ✓ MLP_star_TAR07/20210321-183028/validation
- ☐ MLP_star_TAR01/20210321-183110/train
- ☐ MLP_star_TAR01/20210321-183110/validation
- ☐ MLP_star_TAR07/20210321-184130/train



	Name	Smoothed	Value	Step	Time	Relative
●	MLP_star_TAR07/20210321-183028/train	0.05357	0.05262	49	Sun Mar 21, 13:31:09	39s
○	MLP_star_TAR07/20210321-183028/validation	0.2296	0.2349	49	Sun Mar 21, 13:31:09	39s

Runs

Write a regex to filter runs

- ✓ MLP_star_TAR07/20210321-183028/train
- ✓ MLP_star_TAR07/20210321-183028/validation
- ☐ MLP_star_TAR01/20210321-183110/train
- ☐ MLP_star_TAR01/20210321-183110/validation
- ☐ MLP_star_TAR07/20210321-184130/train

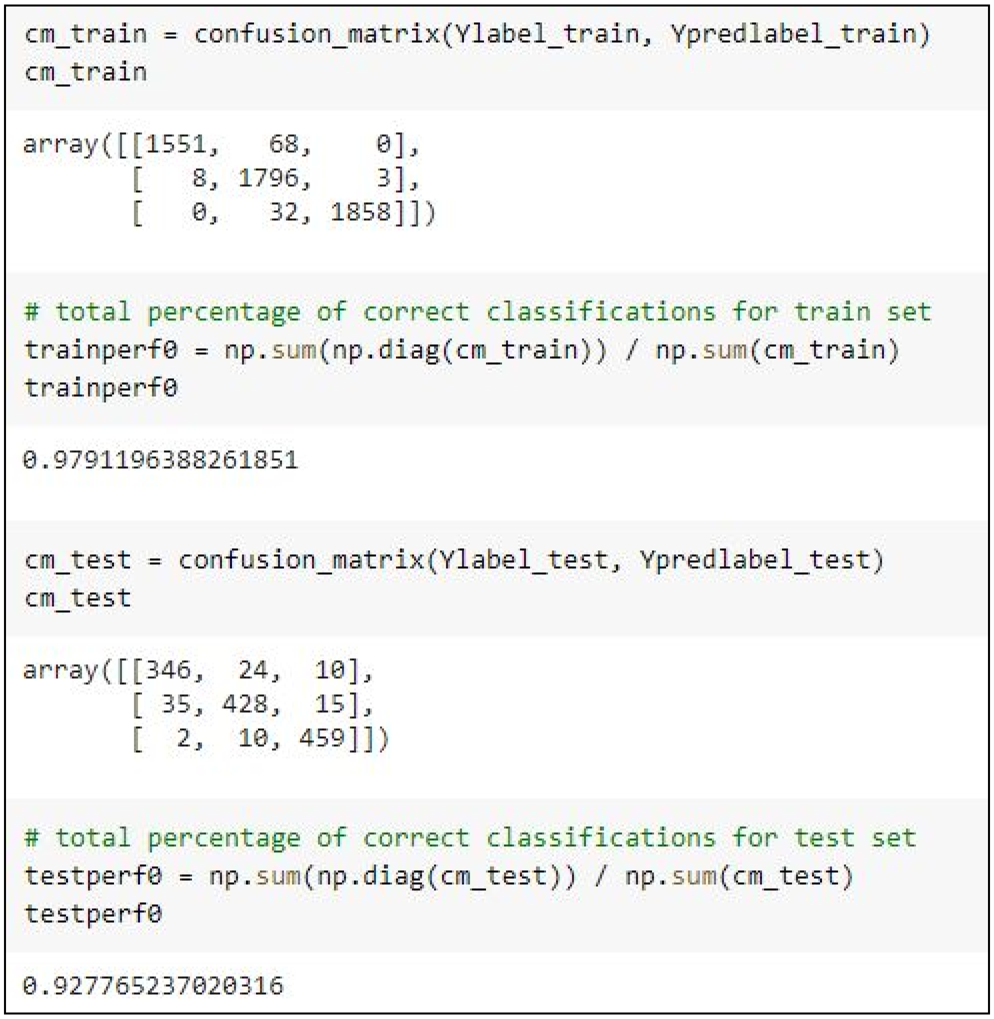
STEP 3: Performance on training and test sets

Using the confusion matrices and total percentages of correct classifications, we are able to evaluate the performance on training and test sets.

INPUT (400 features) → **H** (h0 / h*) → **OUTPUT** (3 font classes) → **softmax** → **Pn**

Where Pn = (% of the correct class)

The following confusion matrix at Tstop & percentage of correct classifications for MLP0:



```
cm_train = confusion_matrix(Ylabel_train, Ypredlabel_train)
cm_train
```

```
array([[1551,   68,    0],
       [   8, 1796,    3],
       [   0,   32, 1858]])
```

```
# total percentage of correct classifications for train set
trainperf0 = np.sum(np.diag(cm_train)) / np.sum(cm_train)
trainperf0
```

```
0.9791196388261851
```

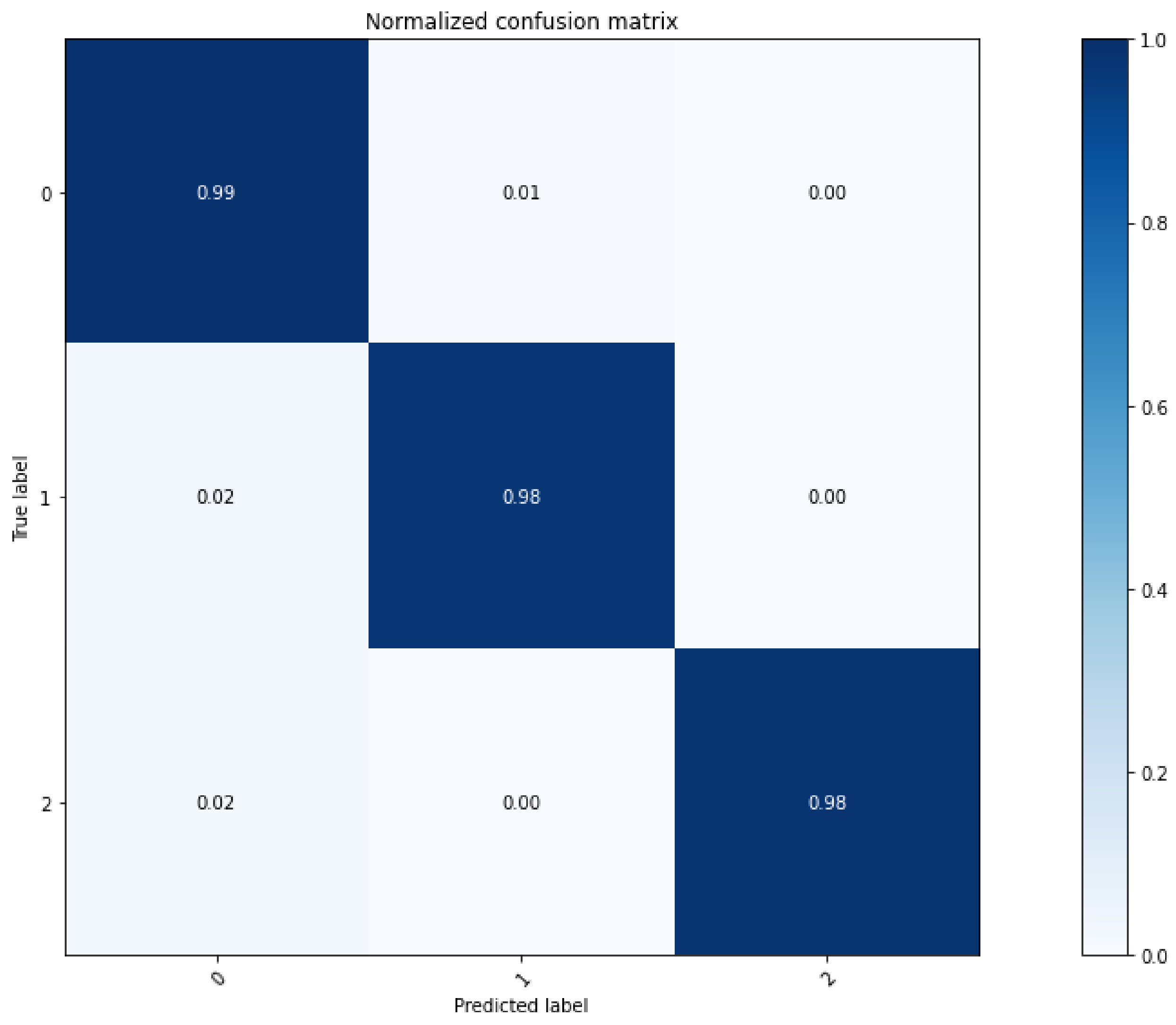
```
cm_test = confusion_matrix(Ylabel_test, Ypredlabel_test)
cm_test
```

```
array([[346,  24,  10],
       [ 35, 428,  15],
       [  2,  10, 459]])
```

```
# total percentage of correct classifications for test set
testperf0 = np.sum(np.diag(cm_test)) / np.sum(cm_test)
testperf0
```

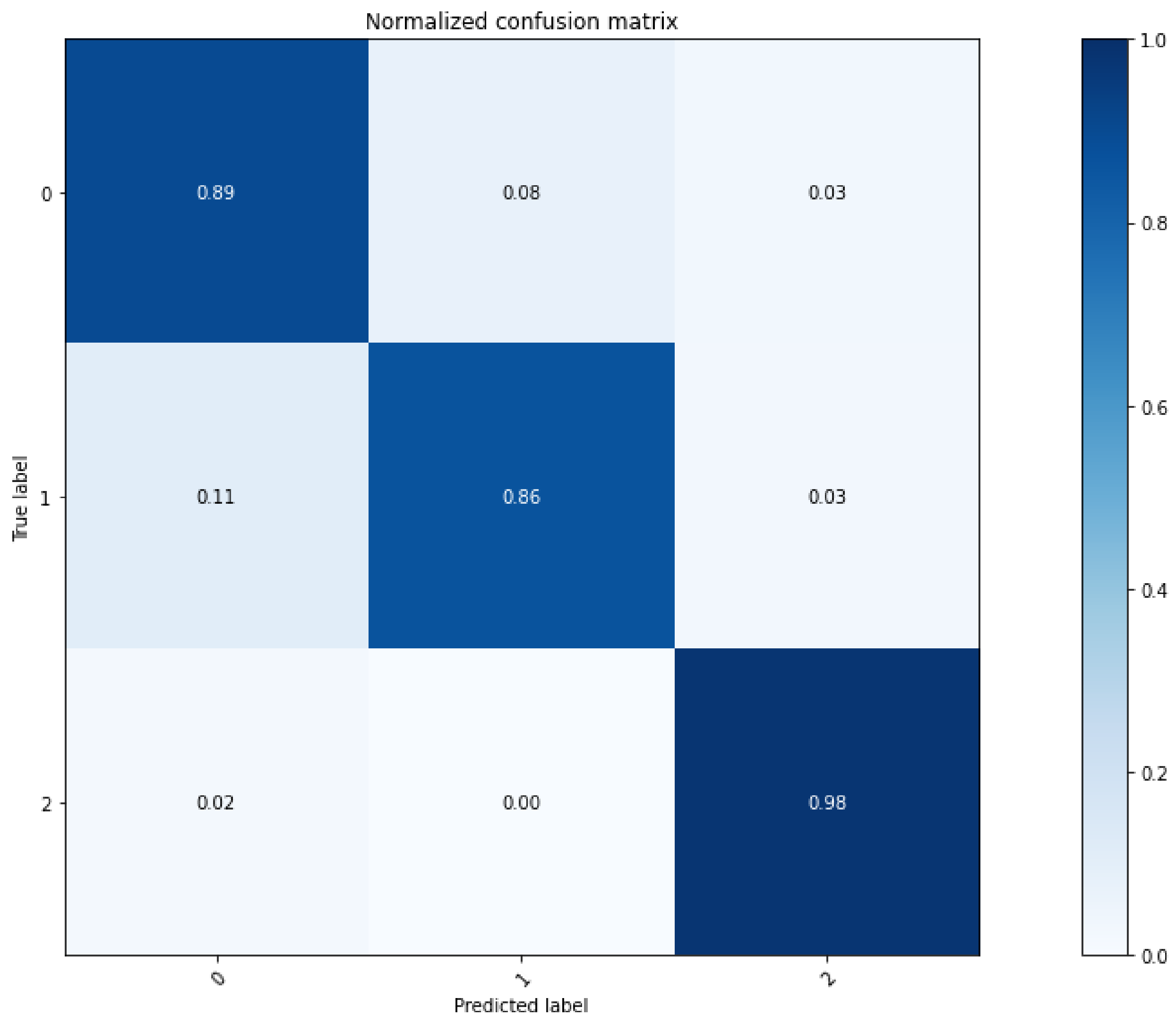
```
0.927765237020316
```

Training set Confusion matrix

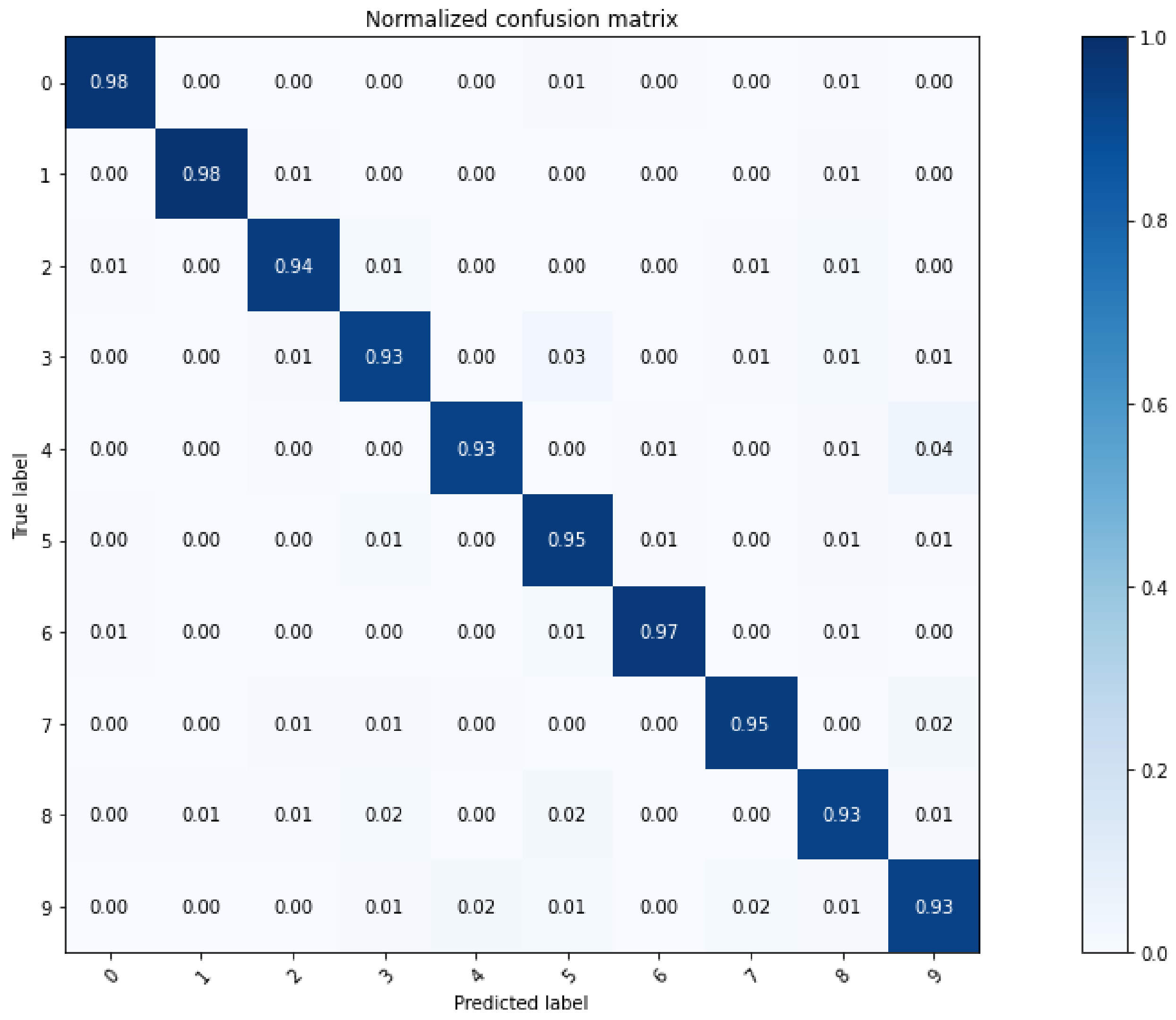


We can see for MLP0, the dark diagonal values are the percentage of correct classifications for the classes and everything else (light blue) is the percentage of times the model was not correct in predicting that class. Overall, our confusion matrix reports that our model performed without much error at all with label prediction accuracies of 99%, 98%, and 98% for our three classes. The confusion matrix for our test set reported moderately high prediction accuracies of 89%, 86%, and 98% for classes CL1, CL2, and CL3.

Test set Confusion matrix



The following confusion matrix at Tstop & percentage of correct classifications for MLP*:

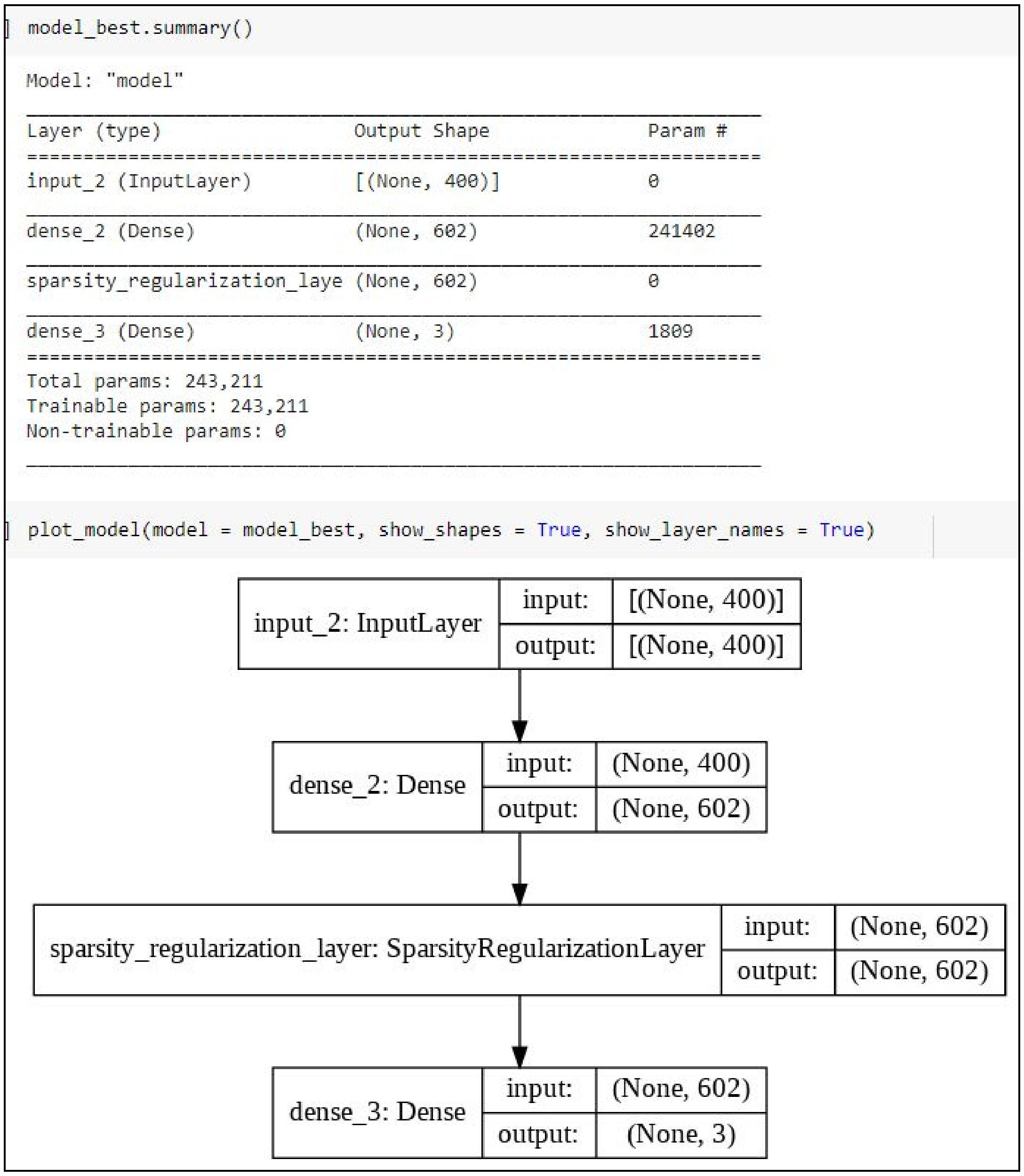


	0	1	2	3	4	5	6	7	8	9
0	0.98	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00
1	0.00	0.98	0.01	0.00	0.00	0.00	0.00	0.00	0.01	0.00
2	0.01	0.00	0.94	0.01	0.00	0.00	0.00	0.01	0.01	0.00
3	0.00	0.00	0.01	0.93	0.00	0.03	0.00	0.01	0.01	0.01
4	0.00	0.00	0.00	0.00	0.93	0.00	0.01	0.00	0.01	0.04
5	0.00	0.00	0.00	0.01	0.00	0.95	0.01	0.00	0.01	0.01
6	0.01	0.00	0.00	0.00	0.00	0.01	0.97	0.00	0.01	0.00
7	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.95	0.00	0.02
8	0.00	0.01	0.01	0.02	0.00	0.02	0.00	0.00	0.93	0.01
9	0.00	0.00	0.00	0.01	0.02	0.01	0.00	0.02	0.01	0.93

We can see for MLP0, the dark diagonal values are the percentage of correct classifications for the classes and everything else (light blue) is the percentage of times the model was not correct in predicting that class. Overall, the performance of the training set is quite high.

STEP 4: Performance at 2 Sparsity Levels

For the hidden layer h* = 602 > k = 400 features, we perform at 2 sparsity levels and select only the best level for h*. Need to note that the sparsity level was set at Tar = 0.01 before. Now we test using tar1 = 0.7 and tar2 = 0.4. Running the MLP* with these levels, with the 400 input and sigmoid activation function we find and define the TAR_best as 0.7, and continue using this MLP* for the rest of the report.



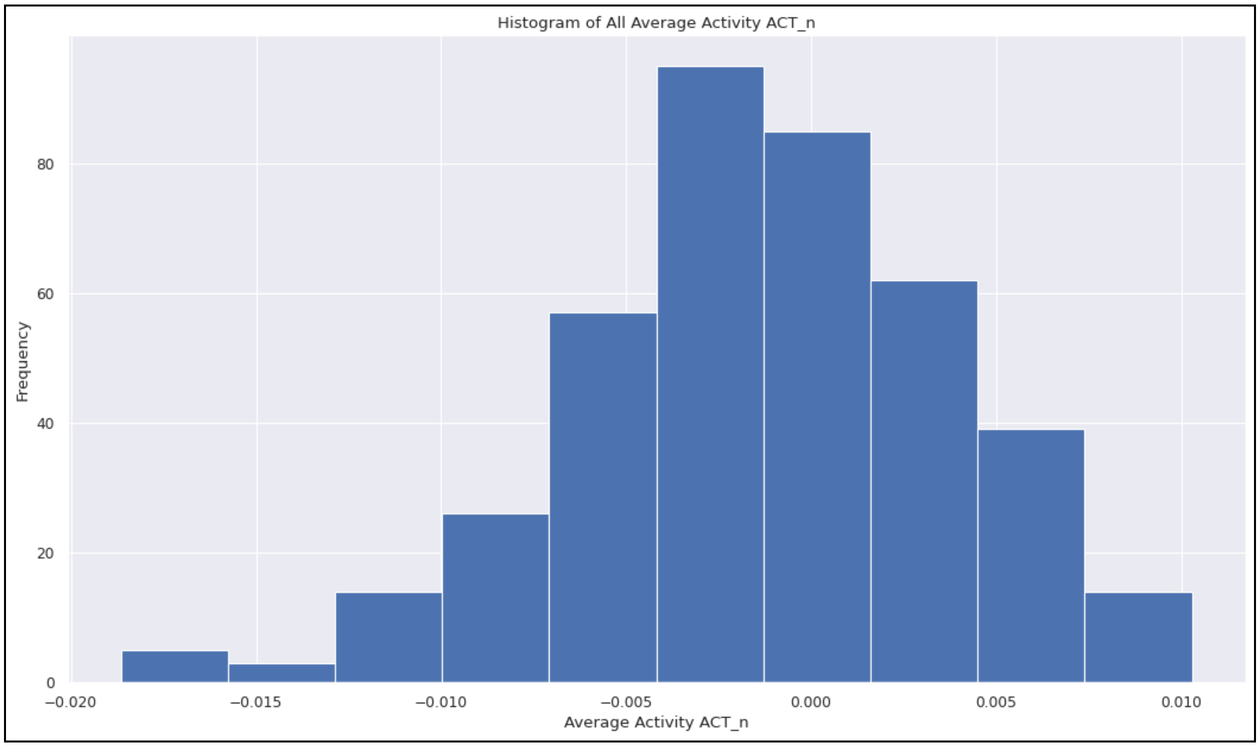
```
model_best.summary()

Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 400)]             0
_________________________________________________________________
dense_2 (Dense)              (None, 602)               241402
_________________________________________________________________
sparsity_regularization_laye (None, 602)               0
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 1809
=================================================================
Total params: 243,211
Trainable params: 243,211
Non-trainable params: 0
_________________________________________________________________

plot_model(model = model_best, show_shapes = True, show_layer_names = True)
```

input_2: InputLayer | input: [(None, 400)] | output: [(None, 400)]
dense_2: Dense | input: (None, 400) | output: (None, 602)
sparsity_regularization_layer: SparsityRegularizationLayer | input: (None, 602) | output: (None, 602)
dense_3: Dense | input: (None, 602) | output: (None, 3)

STEP 5: Average Activity

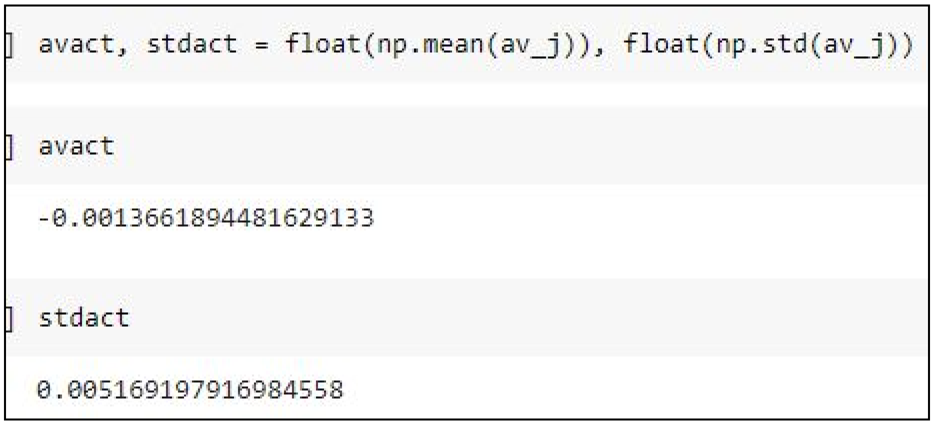
For each input Xn, ACTn = The average activity level over the whole hidden layer. The following data is computed:

```
HIST = histogram of all ACTn values
```



The following is the avact and stdact = The mean and standard deviation of the histogram.

We observe the histogram to have minimal left skew with a mean of average activity level of -0.001366 and a standard deviation of average activity level of 0.005169.



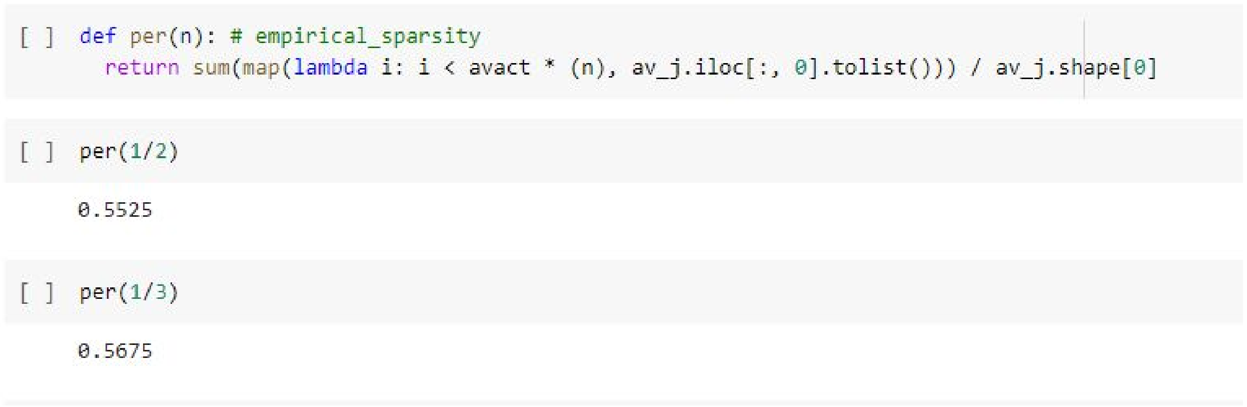
```
] avact, stdact = float(np.mean(av_j)), float(np.std(av_j))

] avact

-0.0013661894481629133

] stdact

0.005169197916984558
```

STEP 6: Empirical Sparsity

The empirical sparsity roughly evaluated by the percentage of ACTn found in step 5 are calculated using the following function:

`per(1/2)` = percentage of ACTn values $< avact/2$ and

`per(1/3)` = percentage of ACTn values $< avact/3$



```
[ ] def per(n): # empirical_sparsity
      return sum(map(lambda i: i < avact * (n), av_j.iloc[:, 0].tolist())) / av_j.shape[0]

[ ] per(1/2)

    0.5525

[ ] per(1/3)

    0.5675
```

We conclude from the histogram and the calculations above, that 55.25% of the ACTn values fall under half of the average activity (*avact*) and 56.75% fall under one third of the average activity.

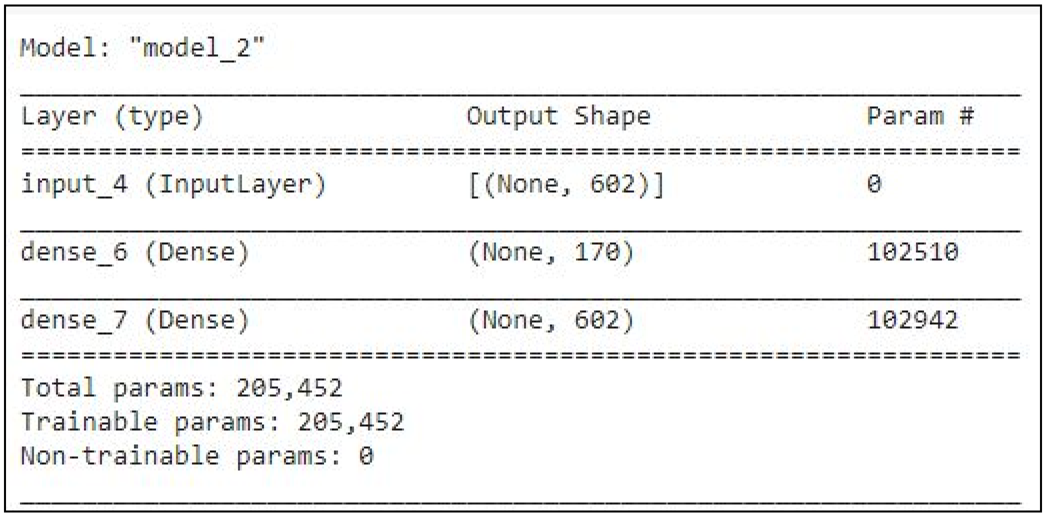
STEP 7: Introducing Zn

From the weights and biases reached at Tstop, we will take a new hidden layer called Zn. The dimensions of Zn are equivalent to h* = 602. Implementing PCA to all the vectors of Zn we compute **s = 170**, where **s** = The number of PCA to explain at 95% variance of what we see on the hidden layer.

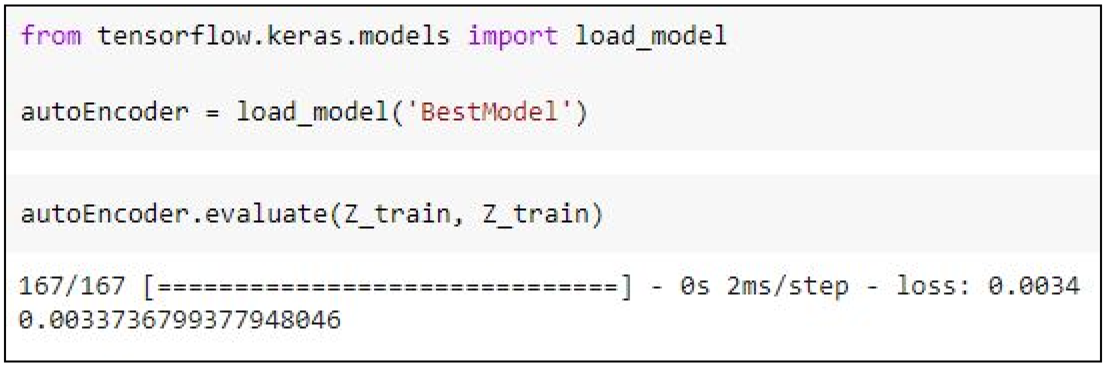
STEP 8: Training Autoencoder (Zn)

Using s from above we can train an autoencoder using Zn as the new input. Note that Zn was the new hidden layer we found earlier, which we will use as the new input layer moving forward. The goal here is to find the weights and basies from Zn to Kn through encoding, then using decoding from Kn to Z'n, where dim(Zn)=dim(Z'n).

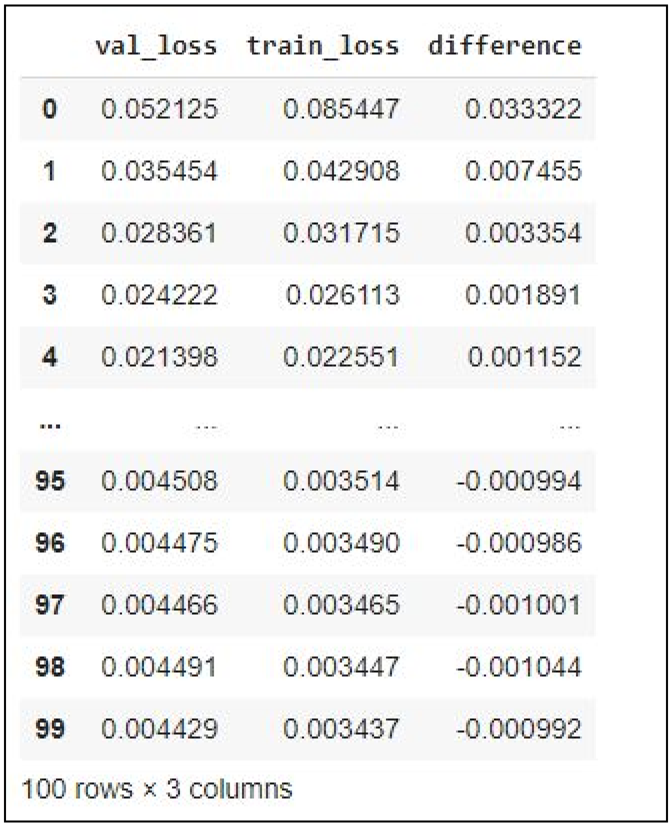
Visually we want something like this: [**Zn** (encode)→ Kn (decode)→ **Z'n**] where dim(Zn) =dim(Z'n) = h* = 602 and dim(Kn) = s


```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 602)]             0
_________________________________________________________________
dense_6 (Dense)              (None, 170)               102510
_________________________________________________________________
dense_7 (Dense)              (None, 602)               102942
=================================================================
Total params: 205,452
Trainable params: 205,452
Non-trainable params: 0
_________________________________________________________________
```

The loss function of this learning is the MSE, average of the error || Zn - Z'n||^2 of all the original data. After running the autoencoder we get the following loss:



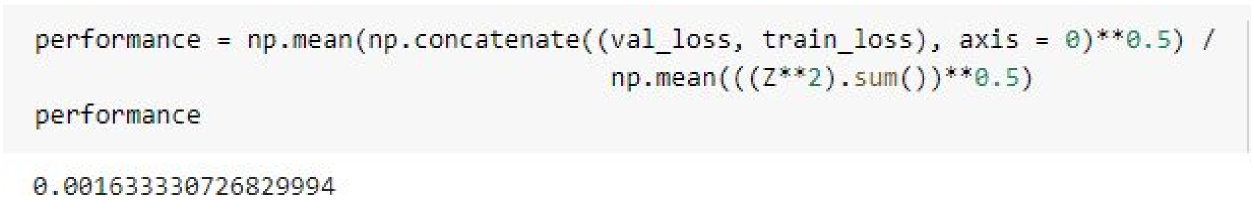
```
from tensorflow.keras.models import load_model

autoEncoder = load_model('BestModel')

autoEncoder.evaluate(Z_train, Z_train)

167/167 [==============================] - 0s 2ms/step - loss: 0.0034
0.0033736799377948046
```



	val_loss	train_loss	difference
0	0.052125	0.085447	0.033322
1	0.035454	0.042908	0.007455
2	0.028361	0.031715	0.003354
3	0.024222	0.026113	0.001891
4	0.021398	0.022551	0.001152
...	...	...	...
95	0.004508	0.003514	-0.000994
96	0.004475	0.003490	-0.000986
97	0.004466	0.003465	-0.001001
98	0.004491	0.003447	-0.001044
99	0.004429	0.003437	-0.000992

100 rows × 3 columns

Evaluating the performance, we calculate the RMSE/ average(||Zn||) to come out to be 0.16%. We want a performance to be less than 5%; ours was a good result.

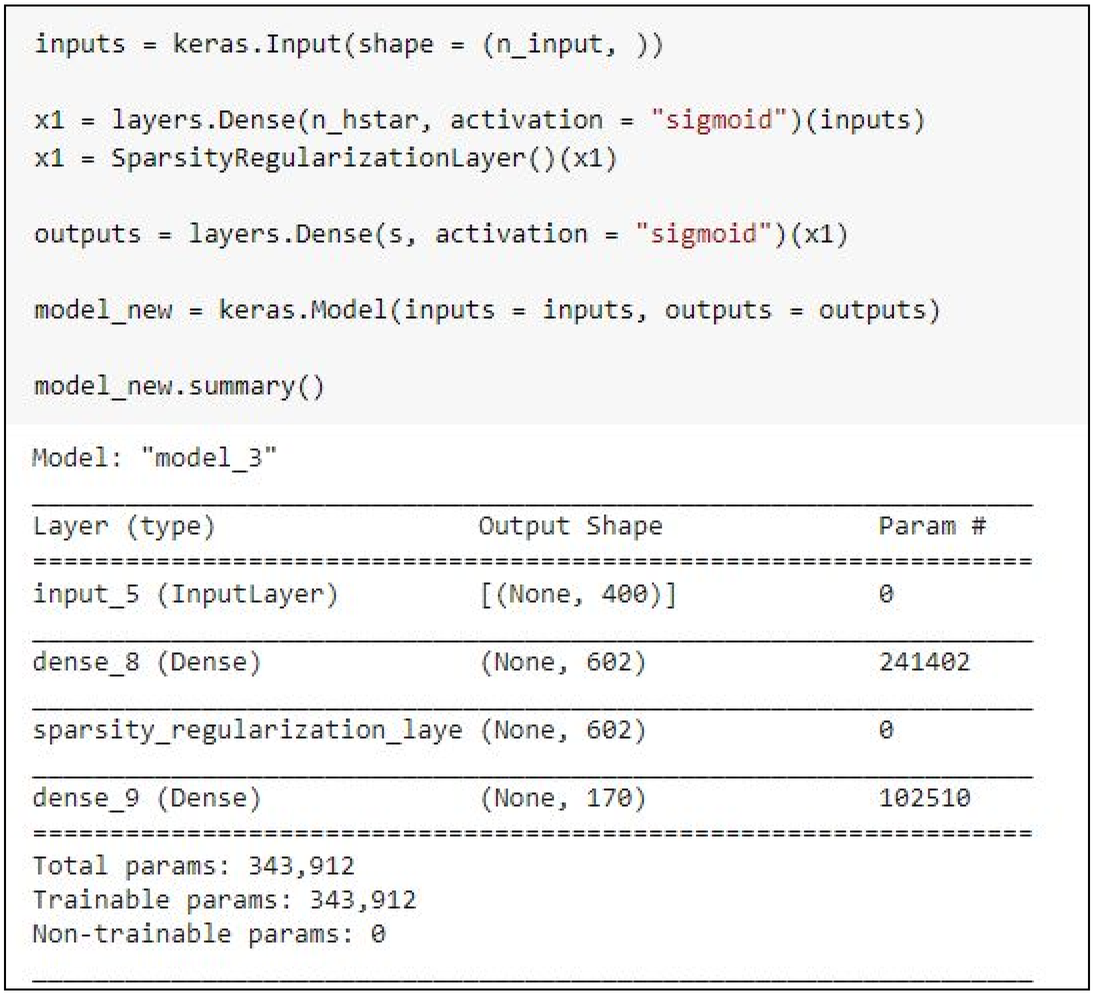


```
performance = np.mean(np.concatenate((val_loss, train_loss), axis = 0)**0.5) /
                                            np.mean(((Z**2).sum())**0.5)
performance

0.001633330726829994
```

STEP 9a: New Input Kn

Moving onto the second step of the deep learning, we will keep the first 2 steps of MLP* (input → h*) we got from earlier, and we will also keep the first 2 layers of the autoencoder (input Zn → Kn) we got in part 8 (Please note that h* = input Zn, to make it simple we will define these two as **H**). we know the weights and biases of both arrows and we stitch them together to get 3 layers of the sequence as: [Input → **H** → Kn] where dim(Input) = k = 400 features, dim(Zn) = h* = 602, and dim(Kn) = s = 170

[400→ 602→ 170] the model summary below observes this.

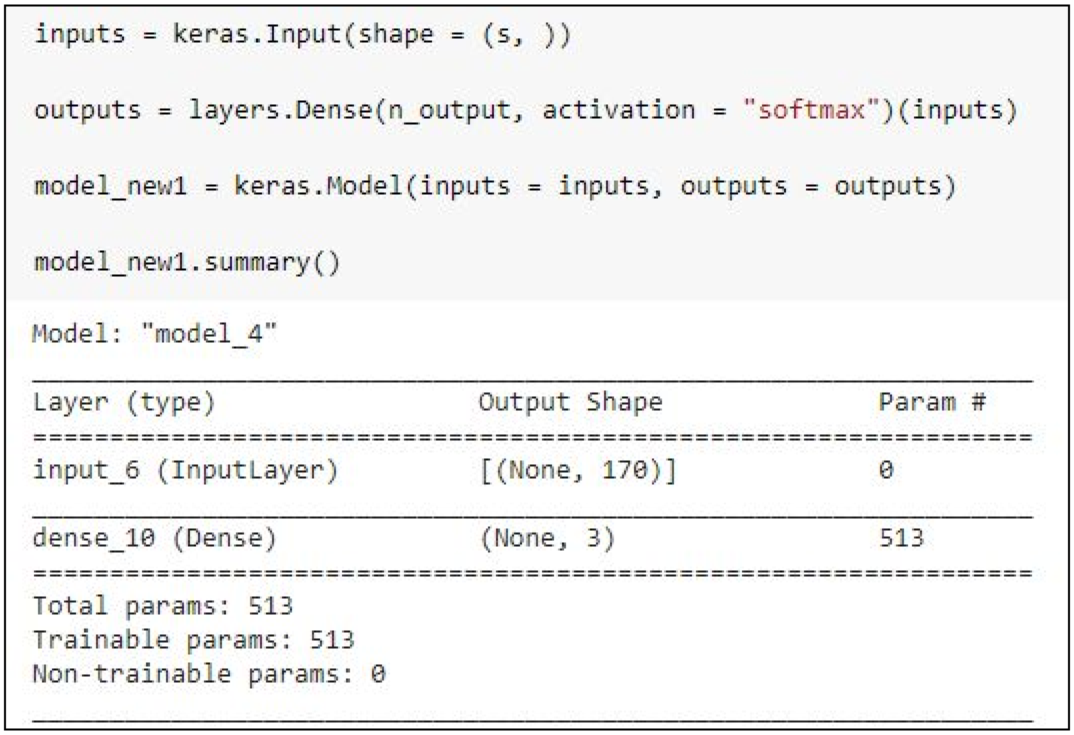


```
inputs = keras.Input(shape = (n_input, ))

x1 = layers.Dense(n_hstar, activation = "sigmoid")(inputs)
x1 = SparsityRegularizationLayer()(x1)

outputs = layers.Dense(s, activation = "sigmoid")(x1)

model_new = keras.Model(inputs = inputs, outputs = outputs)

model_new.summary()
```

```
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 400)]             0
_________________________________________________________________
dense_8 (Dense)              (None, 602)               241402
_________________________________________________________________
sparsity_regularization_laye (None, 602)               0
_________________________________________________________________
dense_9 (Dense)              (None, 170)               102510
=================================================================
Total params: 343,912
Trainable params: 343,912
Non-trainable params: 0
_________________________________________________________________
```

STEP 9b: Short MLP / Long MLP

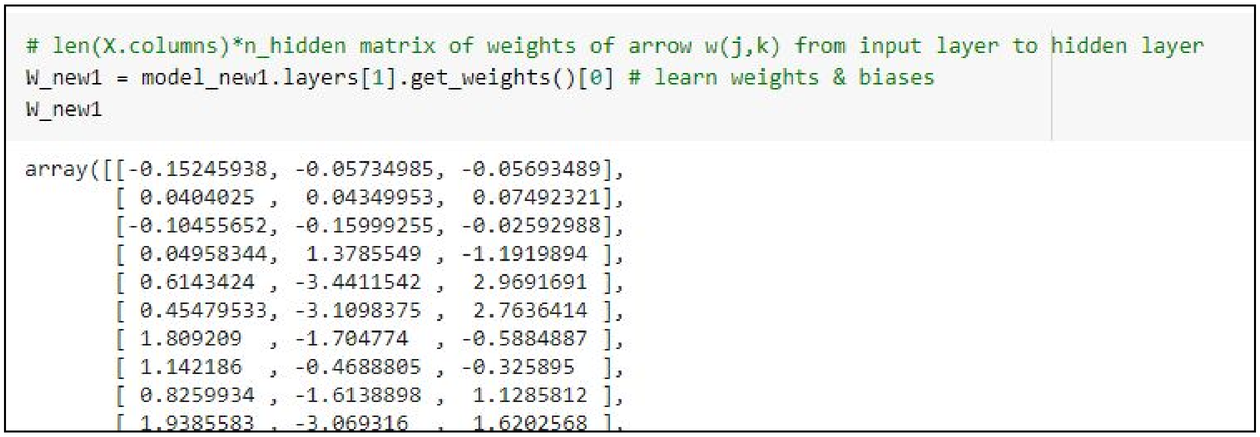
Our Kn is of shape (6645, 170). Compute all the Kn, we use these new inputs Kn with dim(Kn) = s = 170. We train a short MLP, keeping the classification outputs of True Un = [1,0,0], [0,1,0], [0,0,1], when the true class(Xn) = CL1,CL2, CL3 and setting the dim(**OUT**) = 3. Our short MLP will go from S → **OUT** → softmax → Pn (3 percentage probabilities of the 3 classes), note that the **OUT** is unknown here where we hope to learn the new weights and biases; also note that S is the new input Kn we created earlier. This short MLP will use the loss function cross-entropy because we have to compute the three probability and compare them to the true Un. The following model summary is for our [S → **OUT]** using the softmax activation.



```
inputs = keras.Input(shape = (s, ))

outputs = layers.Dense(n_output, activation = "softmax")(inputs)

model_new1 = keras.Model(inputs = inputs, outputs = outputs)

model_new1.summary()
```

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 170)]             0
_________________________________________________________________
dense_10 (Dense)             (None, 3)                 513
=================================================================
Total params: 513
Trainable params: 513
Non-trainable params: 0
_________________________________________________________________
```

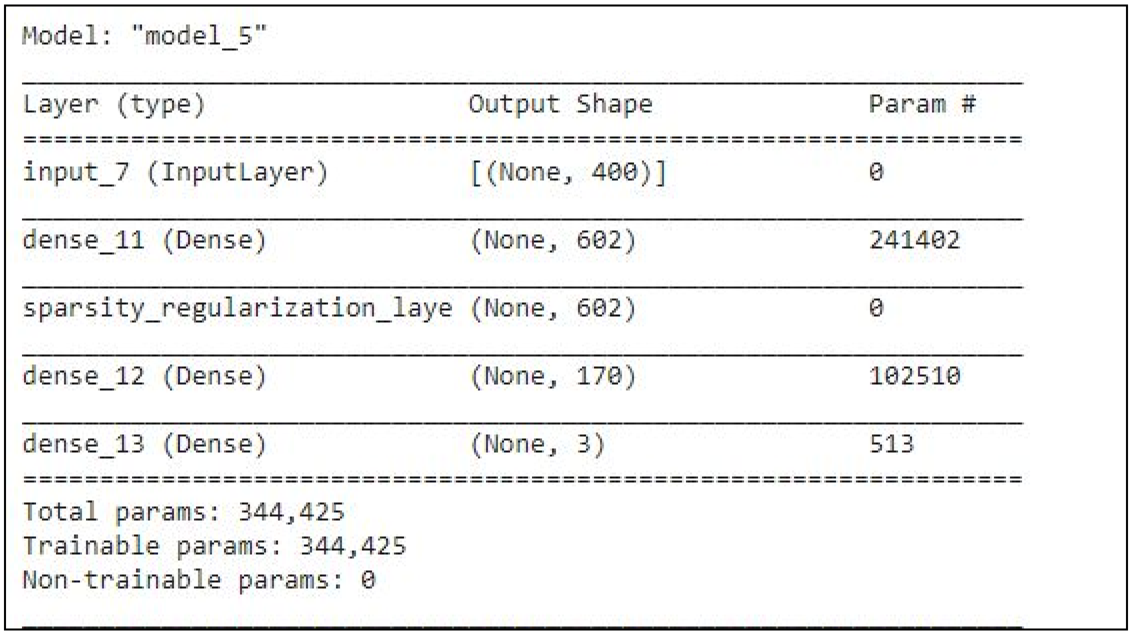
For purposes of this report, we will show the first 10 rows of our 170 new weights for the 3 classes output.



```
# len(X.columns)*n_hidden matrix of weights of arrow w(j,k) from input layer to hidden layer
W_new1 = model_new1.layers[1].get_weights()[0] # learn weights & biases
W_new1
```

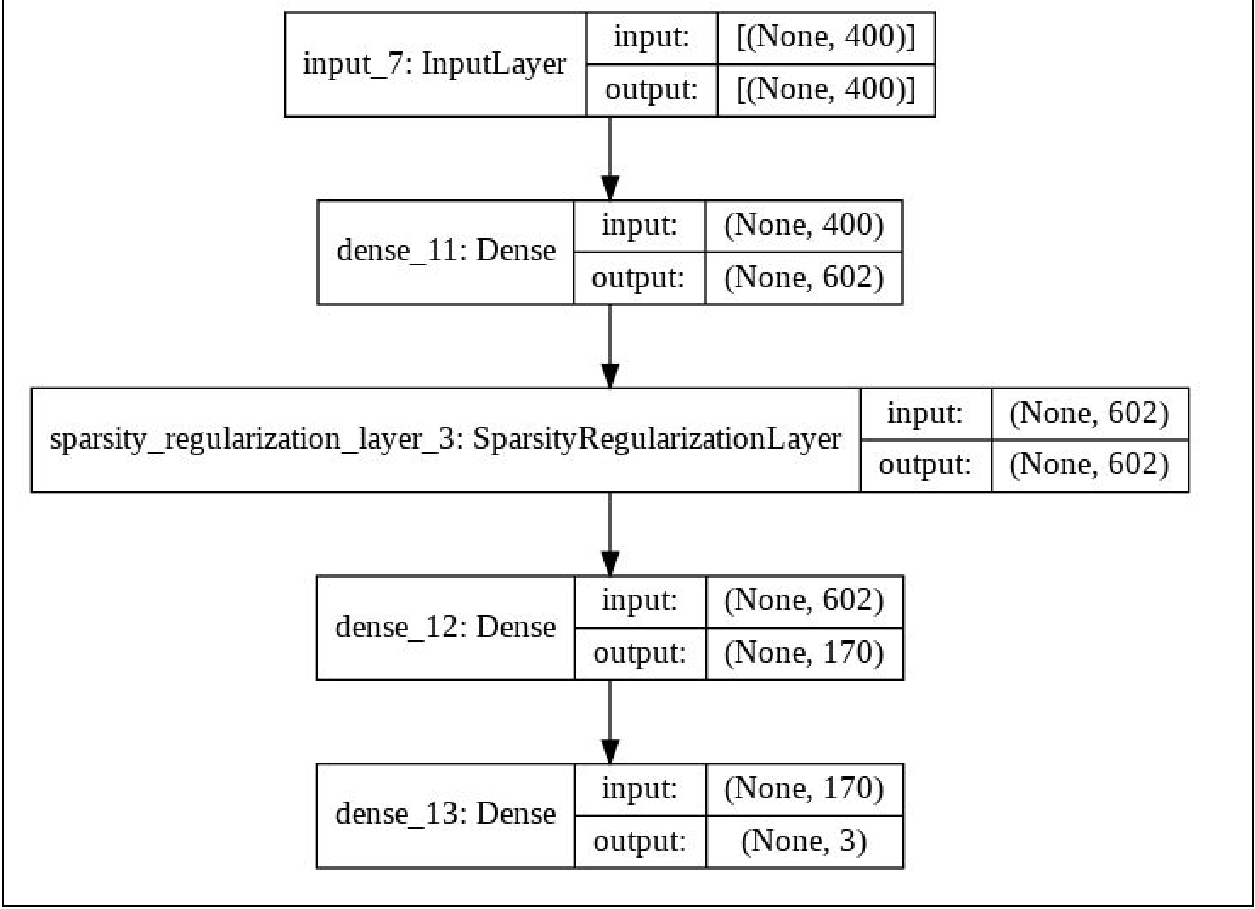
```
array([[-0.15245938, -0.05734985, -0.05693489],
       [ 0.0404025 ,  0.04349953,  0.07492321],
       [-0.10455652, -0.15999255, -0.02592988],
       [ 0.04958344,  1.3785549 , -1.1919894 ],
       [ 0.6143424 , -3.4411542 ,  2.9691691 ],
       [ 0.45479533, -3.1098375 ,  2.7636414 ],
       [ 1.809209  , -1.704774  , -0.5884887 ],
       [ 1.142186  , -0.4688805 , -0.325895  ],
       [ 0.8259934 , -1.6138898 ,  1.1285812 ],
       [ 1.9385583 , -3.069316  ,  1.6202568 ],
```

Moving on, we will build a longer MLP and define it as MLP**. Already defined in part 9a, we have [INPUT → H → S], which we combine with what we have defined in the first half in part 9b [S → OUT]. combining these two we create our long MLP** **[INPUT → H → S → OUT → softmax → Pn]**, where we have a total of 2 hidden layers H and S. Following is the MLP** model summary and model plot for better visualization.



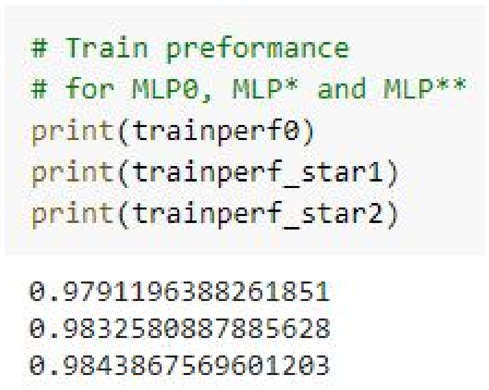
```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, 400)]             0
_________________________________________________________________
dense_11 (Dense)             (None, 602)               241402
_________________________________________________________________
sparsity_regularization_laye (None, 602)               0
_________________________________________________________________
dense_12 (Dense)             (None, 170)               102510
_________________________________________________________________
dense_13 (Dense)             (None, 3)                 513
=================================================================
Total params: 344,425
Trainable params: 344,425
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(model = model_best1, show_shapes = True, show_layer_names = True)
```



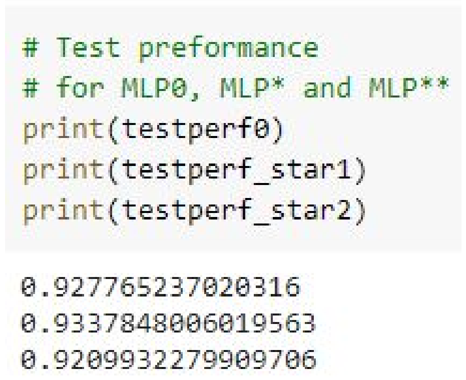
STEP 10: Compare Performances

In our final step in this report, we will compare the performances between our 3 MLPs: MLP0, MLP* nad MLP**. Between both the train and test performances of the 3 MLPs we observe the following results.



```
# Train preformance
# for MLP0, MLP* and MLP**
print(trainperf0)
print(trainperf_star1)
print(trainperf_star2)
```

```
0.9791196388261851
0.9832580887885628
0.9843867569601203
```



```
# Test preformance
# for MLP0, MLP* and MLP**
print(testperf0)
print(testperf_star1)
print(testperf_star2)
```

```
0.927765237020316
0.9337848006019563
0.9209932279909706
```

We observe that the training has improved performance after we adjust our MLPs. Unfortunately, this does not appear to be the same case for our Test set performance at MLP**. Overall our training performance is higher than the test and the results in both sets are high (90% or higher). This indicates we are moving in the right direction.

Conclusion

In this report, we explored a few automatic learning methods for Multiple Layer Perceptron (MLP) classifications to identify the one with the most accurate percentage of class performance (Pn). We are quite familiar with the first one: MLP0 where it had only one hidden layer and its number of hidden layers, h0 = 103, were determined using Principal Component Analysis (PCA) at a 95% explained variance.

Next, we explored MLP*, where it also had one hidden layer, but the dimension of the layer was much larger than its input and output at dim(h*) = 602. This value h* was also obtained by PCA, but it was the sum of the percentage explained variance at 99% for each class. Since the hidden layer was large, we used sparsity learning for this MLP* and identified the best level was at TAR = 0.70

Finally, we explored MLP**, where we combined layers of MLP* and our trained autoencoder in part 8 to create a two hidden layer multiple layer perceptron. Based on the training performances, we can conclude better results the more we adjust or add extensions to the MLP. It would be interesting to explore further MLPs with possibly more than 2 hidden layers, such as 3 or even 4.