Multilayer Perceptron Project Report

MATH 6373 - Deep Learning and Neural Networks

Ravik Chand
Lucas Smith
Huy Nguyen

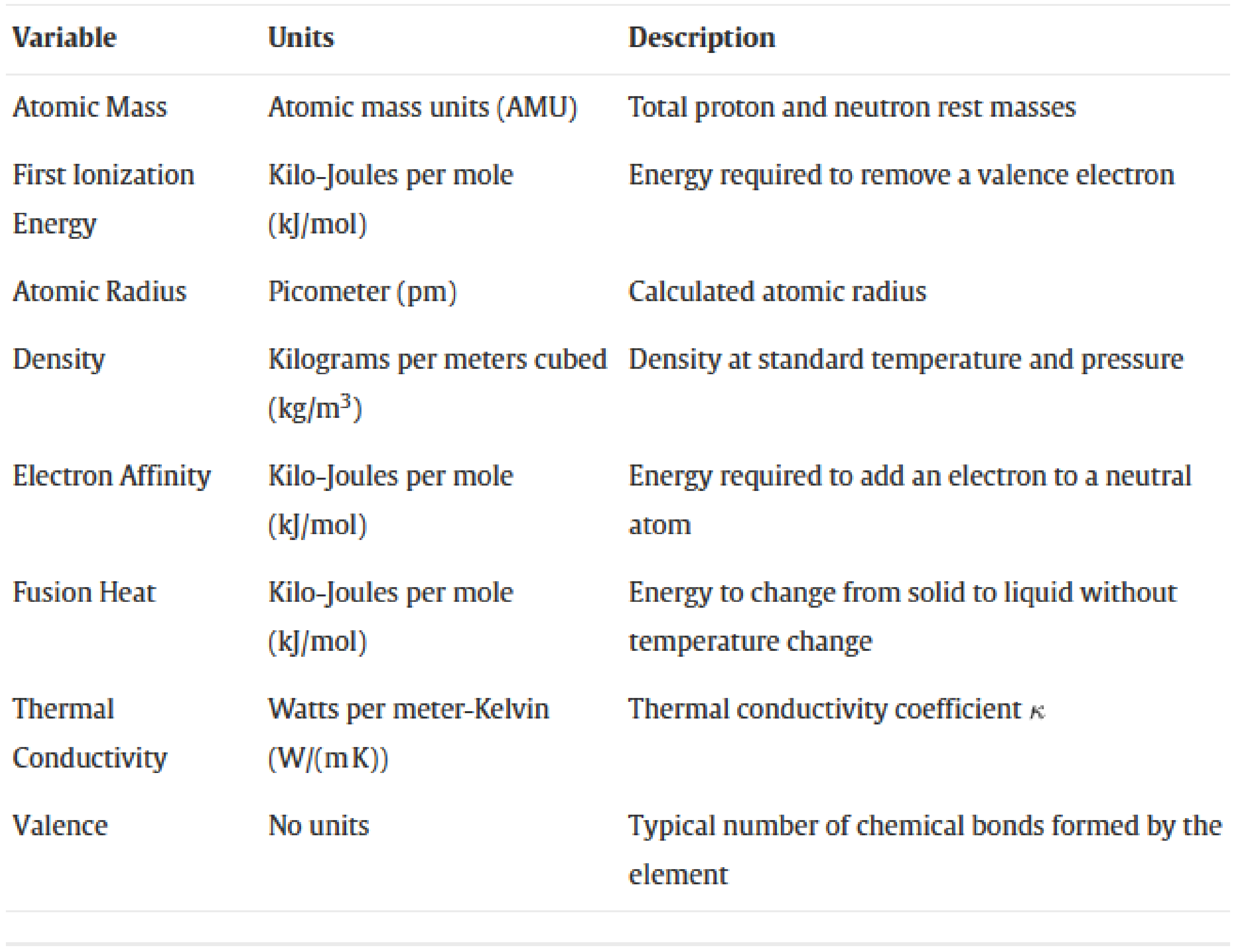
MATH 6373 - Deep Learning and Neural Networks
Multilayer Perceptron Project Report

Ravik Chand
Lucas Smith
Huy Nguyen

Part 0
For this project, our team used The Superconductivity Data Data Set, shared by Hamidieh Kam of the University of Pennsylvania and hosted on the University of California Machine Learning Repository. This dataset contains information about 21,263 instances, *n*, of superconducting materials, their composition, and their respective chemical characteristics.

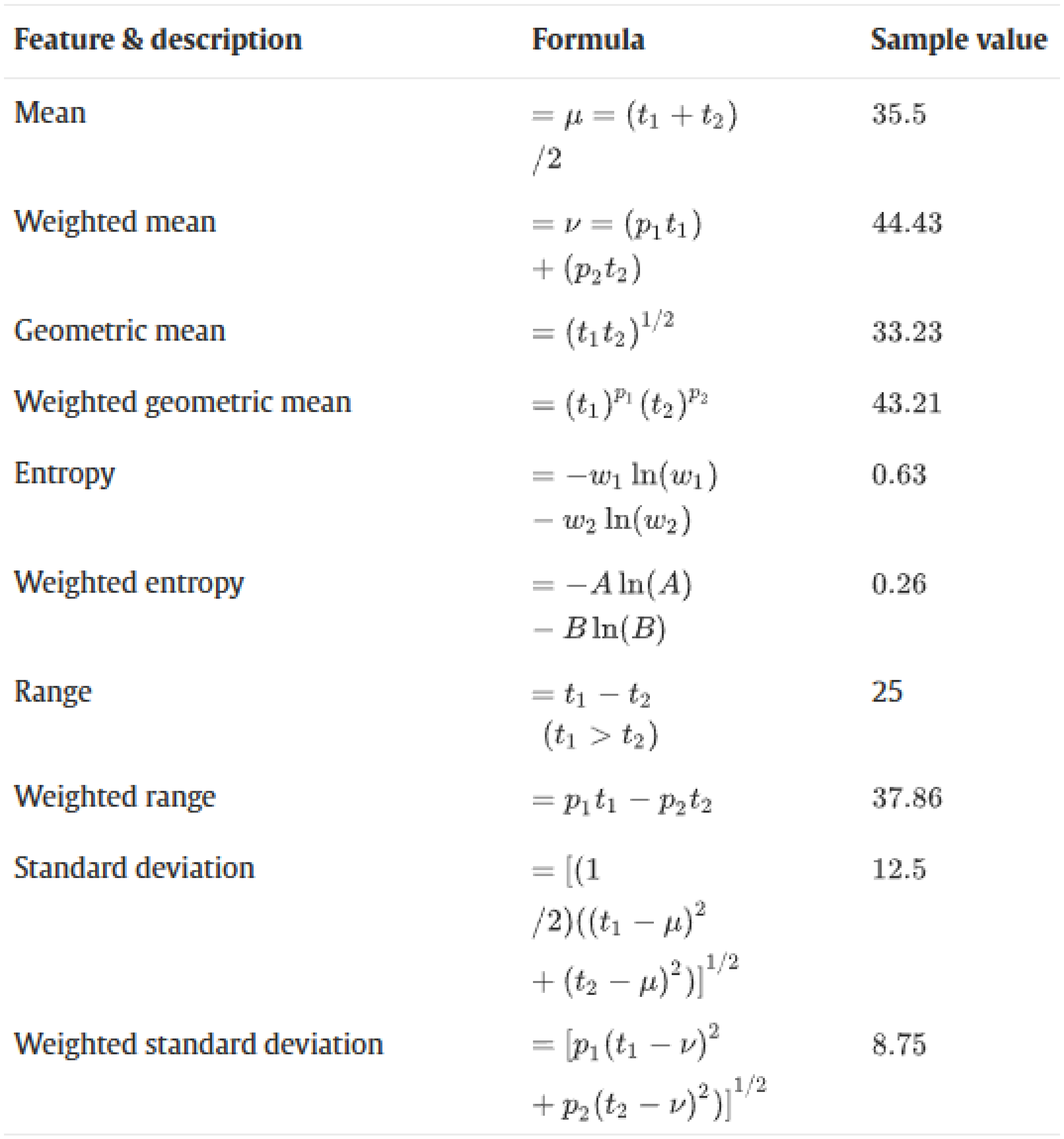
UCI Machine Learning Repository: Superconductivty Data Data Set

In addition to the number of unique elements found in each material, eight variables regarding each material are recorded such as Atomic Mass, First Ionization Energy (FIE), Atomic Radius, Density, Fusion Heat, Thermal Conductivity, and Valence. The definitions of each variable are as below:



Variable	Units	Description
Atomic Mass	Atomic mass units (AMU)	Total proton and neutron rest masses
First Ionization Energy	Kilo-Joules per mole (kJ/mol)	Energy required to remove a valence electron
Atomic Radius	Picometer (pm)	Calculated atomic radius
Density	Kilograms per meters cubed (kg/m^3)	Density at standard temperature and pressure
Electron Affinity	Kilo-Joules per mole (kJ/mol)	Energy required to add an electron to a neutral atom
Fusion Heat	Kilo-Joules per mole (kJ/mol)	Energy to change from solid to liquid without temperature change
Thermal Conductivity	Watts per meter-Kelvin (W/(m K))	Thermal conductivity coefficient κ
Valence	No units	Typical number of chemical bonds formed by the element

From these variables, 80 feature columns, *p*, are derived containing the mean, range, entropy, and standard deviation data of the variables. As an example, regarding the variable "thermal conductivity," the mean feature column represents the arithmetic average of thermal conductivity coefficients of elements found within a material. The weighted mean represents the average adjusted to the proportion of each element found within the material. The geometric mean is an alternative calculation of the average, and likewise, the weighted variant feature column represents the geometric mean calculation with consideration to the proportion of each element found in a material. The range and weighted range represent the difference between the greatest and weakest coefficients. Entropy represents disorder, while standard deviation represents the variation between coefficients of elements within a material. The formulas used to calculate these features, specifically in the case of a superconductor composed of 7Re+1Zr are as below:



Feature & description	Formula	Sample value
Mean	$= \mu = (t_1 + t_2)/2$	35.5
Weighted mean	$= \nu = (p_1 t_1) + (p_2 t_2)$	44.43
Geometric mean	$= (t_1 t_2)^{1/2}$	33.23
Weighted geometric mean	$= (t_1)^{p_1} (t_2)^{p_2}$	43.21
Entropy	$= -w_1 \ln(w_1) - w_2 \ln(w_2)$	0.63
Weighted entropy	$= -A \ln(A) - B \ln(B)$	0.26
Range	$= t_1 - t_2 \; (t_1 > t_2)$	25
Weighted range	$= p_1 t_1 - p_2 t_2$	37.86
Standard deviation	$= [(1/2)((t_1 - \mu)^2 + (t_2 - \mu)^2)]^{1/2}$	12.5
Weighted standard deviation	$= [p_1 (t_1 - \nu)^2 + p_2 (t_2 - \nu)^2)]^{1/2}$	8.75

Ravik Chand
Lucas Smith
Huy Nguyen

colname	**Description**
number_of_elements	Number of unique elements
mean_atomic_mass	Mean atomic mass
wtd_mean_atomic_mass	Weighted mean atomic mass
gmean_atomic_mass	Geometric mean atomic mass
wtd_gmean_atomic_mass	Weighted geometric mean atomic mass
entropy_atomic_mass	Entropy of atomic mass
wtd_entropy_atomic_mass	Weighted entropy of atomic mass
range_atomic_mass	Range of atomic mass
wtd_range_atomic_mass	Weighted range of atomic mass
std_atomic_mass	Standard deviation of atomic mass
wtd_std_atomic_mass	Weighted standard deviation of atomic mass
mean_fie	Mean first ionization energy
wtd_mean_fie	Weighted mean first ionization energy
gmean_fie	Geometric mean first ionization energy
wtd_gmean_fie	Weighted geometric mean first ionization energy
entropy_fie	Entropy of first ionization energy
wtd_entropy_fie	Weighted entropy of first ionization energy
range_fie	Range of first ionization energy
wtd_range_fie	Weighted range of first ionization energy
std_fie	Standard deviation of first ionization energy
wtd_std_fie	Weighted standard deviation of first ionization energy
mean_atomic_radius	Mean atomic radius
wtd_mean_atomic_radius	Weighted mean atomic radius
gmean_atomic_radius	Geometric mean atomic radius
wtd_gmean_atomic_radius	Weighted geometric mean atomic radius
entropy_atomic_radius	Entropy of atomic radius
wtd_entropy_atomic_radius	Weighted entropy of atomic radius
range_atomic_radius	Range of atomic radius
wtd_range_atomic_radius	Weighted range of atomic radius
std_atomic_radius	Standard deviation of atomic radius
wtd_std_atomic_radius	Weighted standard deviation of atomic radius
mean_Density	Mean density

Ravik Chand
Lucas Smith
Huy Nguyen

wtd_mean_Density	Weighted mean density
gmean_Density	Geometric mean density
wtd_gmean_Density	Weighted geometric mean density
entropy_Density	Entropy of density
wtd_entropy_Density	Weighted entropy of density
range_Density	Range of density
wtd_range_Density	Weighted range of density
std_Density	Standard deviation of density
wtd_std_Density	Weighted standard deviation of density
mean_ElectronAffinity	Mean electron affinity
wtd_mean_ElectronAffinity	Weighted mean electron affinity
gmean_ElectronAffinity	Geometric mean electron affinity
wtd_gmean_ElectronAffinity	Weighted geometric mean electron affinity
entropy_ElectronAffinity	Entropy of electron affinity
wtd_entropy_ElectronAffinity	Weighted entropy of electron affinity
range_ElectronAffinity	Range of electron affinity
wtd_range_ElectronAffinity	Weighted range of electron affinity
std_ElectronAffinity	Standard deviation of electron affinity
wtd_std_ElectronAffinity	Weighted standard deviation of electron affinity
mean_FusionHeat	Mean fusion heat
wtd_mean_FusionHeat	Weighted mean fusion heat
gmean_FusionHeat	Geometric mean fusion heat
wtd_gmean_FusionHeat	Weighted geometric mean fusion heat
entropy_FusionHeat	Entropy of fusion heat
wtd_entropy_FusionHeat	Weighted entropy of fusion heat
range_FusionHeat	Range of fusion heat
wtd_range_FusionHeat	Weighted range of fusion heat
std_FusionHeat	Standard deviation of fusion heat
wtd_std_FusionHeat	Weighted standard deviation of fusion heat
mean_ThermalConductivity	Mean thermal conductivity
wtd_mean_ThermalConductivity	Weighted mean thermal conductivity
gmean_ThermalConductivity	Geometric mean thermal conductivity
wtd_gmean_ThermalConductivity	Weighted geometric mean thermal conductivity

Ravik Chand
Lucas Smith
Huy Nguyen

entropy_ThermalConductivity	Entropy of thermal conductivity
wtd_entropy_ThermalConductivity	Weighted entropy of thermal conductivity
range_ThermalConductivity	Range of thermal conductivity
wtd_range_ThermalConductivity	Weighted range of thermal conductivity
std_ThermalConductivity	Standard deviation of thermal conductivity
wtd_std_ThermalConductivity	Weighted standard deviation of thermal conductivity
mean_Valence	Mean valence
wtd_mean_Valence	Weighted mean valence
gmean_Valence	Geometric mean valence
wtd_gmean_Valence	Weighted geometric mean valence
entropy_Valence	Entropy of valence
wtd_entropy_Valence	Weighted entropy of valence
range_Valence	Range of valence
wtd_range_Valence	Weighted range of valence
std_Valence	Standard deviation of valence
wtd_std_Valence	Weighted standard deviation of valence
critical_temp	Critical temperature in Kelvin

The remaining column denotes the critical temperature required to achieve a superconductive state, which will be used to define our classes as below.

Critical Temp.	Class	Cases
3.32 <	Class 1	3541
3.32 — 8.40	Class 2	3524
8.40 — 20.00	Class 3	3493
20.00 — 40.57	Class 4	3459
40.57 — 79.00	Class 5	3411
> 79.00	Class 6	3621

The objective of this classification activity is to identify relationships between the elemental variables of materials and the critical temperatures required to achieve a superconductive state. This may help to guide research towards creating superconductive materials which are viable at ambient temperatures.

MATH 6373 - Deep Learning and Neural Networks
Multilayer Perceptron Project Report

Ravik Chand
Lucas Smith
Huy Nguyen

Part 1

The data set was split into 6 dataframes *e1, e2, e3, e4, e5,* and *e6* such that each data frame contained a single class, *j*, as outlined in the above class definition table. Because our classes were of a similar size, SMOTE or other such oversampling techniques were not required to balance the data set. The "critical_temp" column was removed from dataframes *ej*. A new column, "label," was then added containing class identification as *CLj* to each data frame *ej*, for *j* 1 through *k* = 6. The train_test_split function was applied to create training sets, *train_ej,* consisting of 80% of observations, and test sets, *test_ej,* containing 20% of observations from each data frame *ej.* The individual *train_ej* data frames were then concatenated into the data frame *train_data,* and likewise the individual *test_ej* data frames formed the data frame *test_data.* The get_dummies function from the Pandas package was applied to the label column for the *train_data* and *test_data* to create the data frames *one_hot_train* and *one_hot_test* which contained the one-hot encoded categorical variable of class identification. The label columns in both the *train_data* and *test_data* were removed, and columns from the corresponding one-hot encoded frames were added in their respective places. The data frames *x_train* and *x_test* were derived from the test and train data frames by removing the one-hot encoded class information. This class information was isolated to create the *y_train* and *y_test* data frames.

Part 2

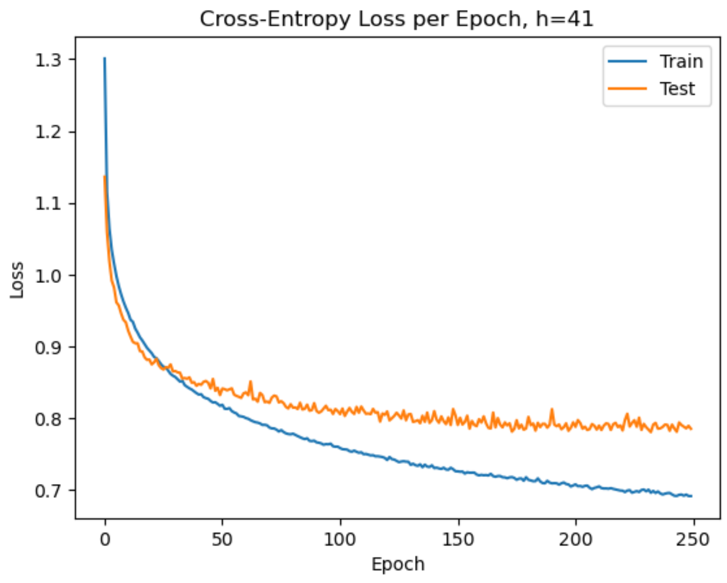
A sequential model from the TensorFlow package was used to create a multilayer perceptron consisting of three layers. The first layer, L1, accepted an input vector *Xn*. This was followed by a hidden layer H. The third layer, L3, served as an output layer. The vector of *k* outputs then passed through the softmax function which translated the values into a probability vector *Pn* in which *Pn(j)* was expressed as a value between 0 and 1 that described the likelihood of case *n* belonging to a particular class *j*. Case n was determined to belong to the class which expressed the strongest probability. *Pn(j)* was calculated as the exponential of *j* divided by the sum of all exponentials of j for a particular case *n*:

$$Pn(j) = e^{j} / \sum_{j=1}^{k} e^{j}$$

Cross entropy (CRE), is a measure of the difference between two probability distributions, in our case the true *CLj, y(n),* or *y_train(n)* and *y_test(n)* versus the *CLj* predicted by *Pn*. This serves as the loss function in our models and is defined by the below formula:

$$CRE[y(n), Pn] = -\ y(n)\ *\ log(Pn)$$

The cross entropy loss is minimized during the training process to adjust the weights and biases of the model in order to improve its performance. The average cross entropy for the training set, *train_avCRE*, and test set, *test_avCRE*, were defined. The ADAM optimizer was specified as well as the RELU response function. The initial weights were randomized, the number of epochs, *m*, was set to 150. The batch size of 152 was determined by the count of *n* cases in *train_data* divided by 100. The initial value of *h*, the count of neurons in the hidden layer, was set as half the count of features, *p*, or 41. Because the training loss showed signs of continuing decline beyond 150 epochs, the number of epochs was extended to 250. The performance of the model as determined by cross entropy loss is displayed in the following graph:



This incurred an additional minute of compute time per 50 epochs or about 0.46 seconds per epoch. Though the training loss continued to decrease beyond 250 epochs, the test loss leveled off leading us to conclude that 250 epochs was the optimal amount, *m**. The below confusion matrix was generated using the *test_data* by MLP* which had a number of epochs equal to *m**.

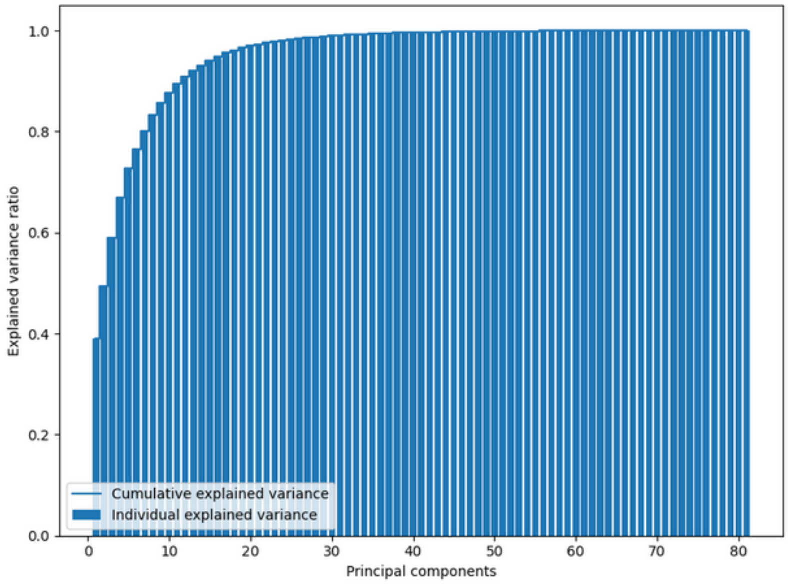
	CL1	**CL2**	**CL3**	**CL4**	**CL5**	**CL6**
CL1	80.40	16.23	3.06	0.15	0.00	0.15
CL2	20.99	69.48	8.59	0.79	0.16	0.00
CL3	6.71	18.33	65.96	8.18	0.65	0.16
CL4	1.17	6.23	22.21	60.65	8.44	1.30
CL5	0.28	0.14	5.83	18.86	58.67	16.23
CL6	0.00	0.12	1.09	3.98	22.92	71.89

CL1, materials with critical temperatures less than 3.32°K, exhibited the greatest prediction accuracy while CL5, materials with critical temperatures greater than 40.57°K but less than 79.00°K, had the weakest.

Ravik Chand
Lucas Smith
Huy Nguyen

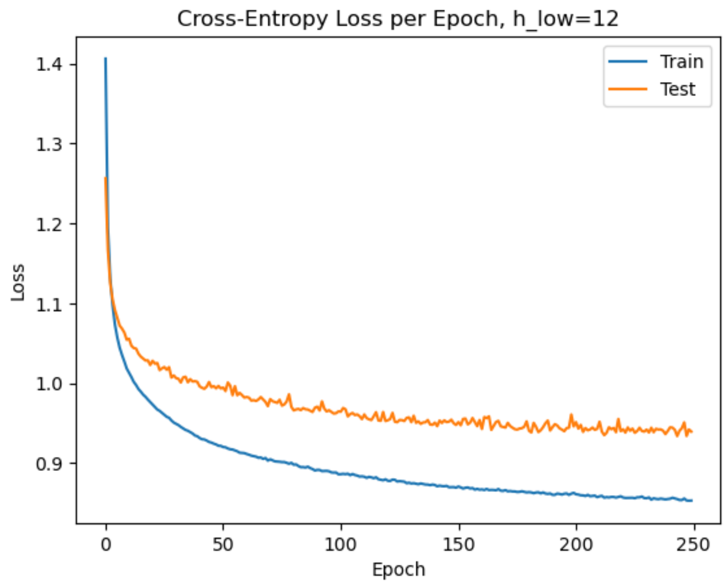
Part 3

Principal component analysis was performed on the training data, *x_train,* to identify the number of features which could explain 90% of variance. From this, 12 principal components were determined.



However, prediction accuracy given this reduced count of features was roughly 6-8% less than the complete feature count, and as such, we maintained usage of all 80 features moving forward. Another MLP was created, MLP_{LOW}, which was given identical attributes to MLP* with the exception of the number of neurons in layer H. While the hidden layer of MLP* was equipped with 41 neurons, MLP_{LOW}'s hidden layer was reduced to 12, h_{LOW}. The results of MLP_{LOW} are as below with the confusion matrix pertaining to the *test_data*:

Ravik Chand
Lucas Smith
Huy Nguyen

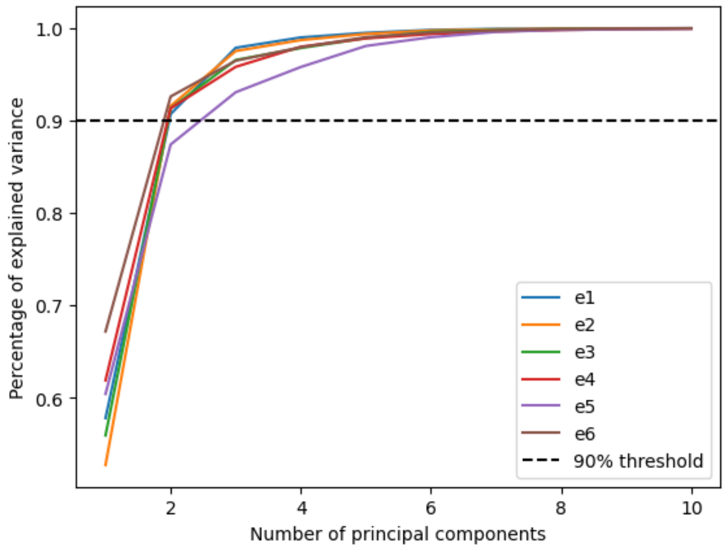


	CL1	CL2	CL3	CL4	CL5	CL6
CL1	72.18	24.16	3.07	0.15	0.15	0.29
CL2	27.01	58.14	13.11	1.26	0.47	0.00
CL3	5.52	20.07	66.30	7.73	0.00	0.37
CL4	1.46	6.89	23.18	60.40	7.42	0.66
CL5	0.36	0.95	6.27	17.63	50.89	23.91
CL6	0.13	0.40	0.93	4.77	25.60	68.17

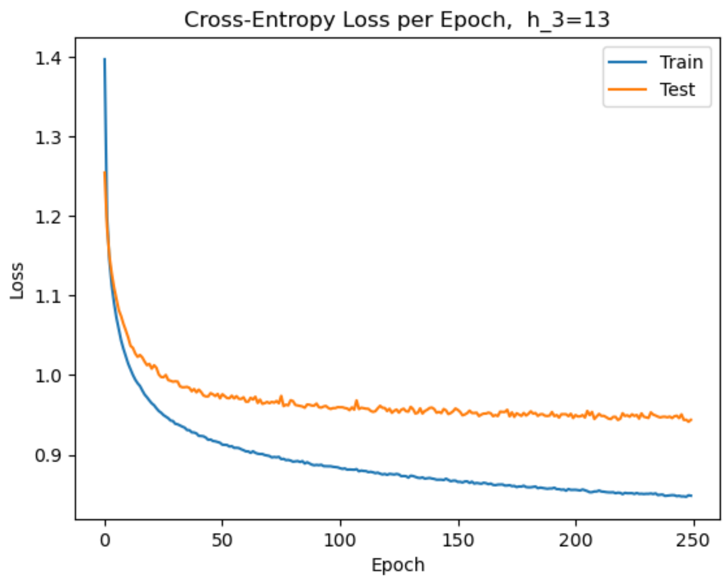
Part 4

PCA was implemented again, however, separately for each homogenous class data frame *ej*. The count of principal components that captured 90% of variation for each data frame *ej* was determined to be 2, with the exception of *e5* which had 3 principal components. The sum of principal components, equal to 13, was used to define h_{PCA}. Using h_{PCA} as the count of neurons in its hidden layer, MLP_{PCA} was created.

Ravik Chand
Lucas Smith
Huy Nguyen



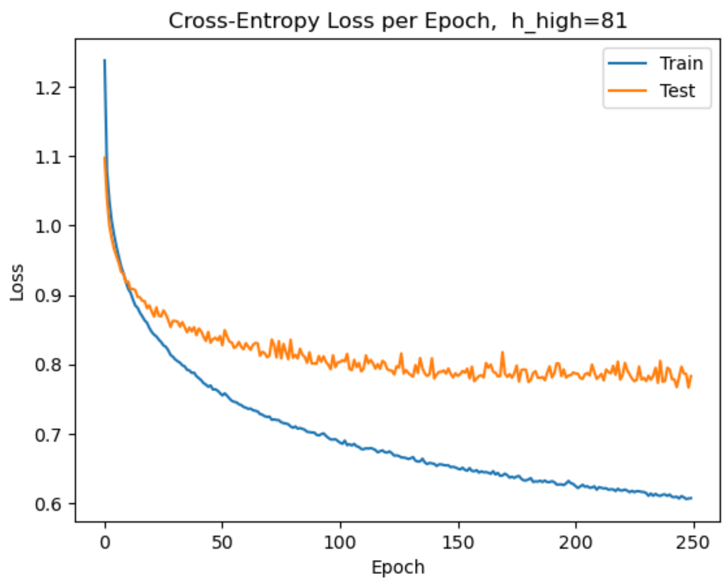
The performance of MLP$_3$ and corresponding confusion matrix corresponding to the *test_data* is as follows:



Ravik Chand
Lucas Smith
Huy Nguyen

	CL1	CL2	CL3	CL4	CL5	CL6
CL1	72.98	22.20	4.04	0.62	0.16	0.00
CL2	27.56	58.15	13.28	0.72	0.29	0.00
CL3	6.27	18.12	63.94	10.63	0.35	0.70
CL4	1.01	7.07	21.93	61.62	7.36	1.01
CL5	0.38	0.51	6.61	19.82	52.48	20.20
CL6	0.24	0.24	1.22	4.74	26.03	67.52

Because h_3 was less than the initial value of h used in MLP*, we opted to use the value p to define h_{HIGH}. The model MLP_{HIGH} was then created with a count of neurons h_{HIGH}, which performed as below along with the confusion matrix for *test_data*:



	CL1	CL2	CL3	CL4	CL5	CL6
CL1	80.23	17.94	1.69	0.14	0.00	0.00
CL2	18.03	71.77	9.01	0.85	0.34	0.00
CL3	3.78	16.79	67.93	10.29	0.76	0.45
CL4	1.23	6.01	19.72	66.41	6.32	0.31
CL5	0.27	0.67	6.48	20.11	57.76	14.71
CL6	0.00	0.12	1.04	4.39	23.90	70.55

Ravik Chand
Lucas Smith
Huy Nguyen

The following table presents the performance of each MLP:

MLP	*h*	*m*	TRAIN CRE	TEST CRE	TEST Acc. (%)	Compute Time (sec.)
MLP_INIT	41	175	4.1170	0.0002	67.67	37.15
MLP_LOW	12	225	4.1170	0.0002	62.21	34.95
MLP_PCA	13	160	4.5746	0.0002	62.54	35.90
MLP_HIGH	81	160	4.5746	0.0002	69.05	32.92

Based on the table above, we picked MLP_HIGH as our MLP* due to its test accuracy and compute time.

Part 5

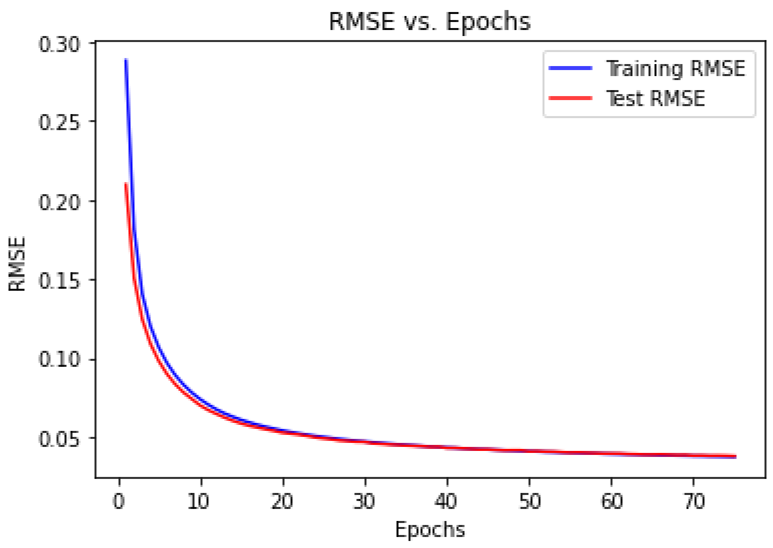
Sparsity learning is a type of machine learning technique that involves reducing the number of features or weights in a model to improve its efficiency and accuracy. In sparsity learning, a model is trained to identify the most important features or weights in the input data and discard the rest. This approach is particularly useful in situations where the input data has a large number of features or weights that may not be relevant or useful for the task at hand.

Sparsity learning can be achieved through various methods, such as L1 regularization, which adds a penalty term to the loss function that encourages the weights of less important features to be set to zero. Another method is to use feature selection techniques, such as correlation-based feature selection, which identifies features that are highly correlated with the output variable and removes the rest.

Sparsity learning has several advantages, including faster training times, improved generalization performance, and increased interpretability of the model. It is commonly used in applications such as natural language processing, image and signal processing, and recommendation systems.

The **get_weight()** method was used to obtain the thresholds, *W*, and thresholds, *B*, for MLP_{HIGH}. A new model, hidden_model4, was created with only an input layer and the hidden layer referred from MLP_{HIGH} which could be used to define the data frames of activity vectors Zn_train and Zn_test for the x_train and x_test respectively. These new variables were used to train an autoencoder which could decode the activity vectors. Additionally, a Root Mean Squared Error method was defined to serve as the loss metric for automatic learning, and a Sparse Loss method as the loss function for sparsity learning. The sparsity target was defined as 10% and sparsity weight set to 0.001. The autoencoder was trained on the Zn_train data frame and validated using the Zn_test frame. The RMSE for the training and testing was plotted per epoch in the below graph:

Ravik Chand
Lucas Smith
Huy Nguyen



Based on this graph, 70 was selected as our optimal number of epochs, m_{AEC}. The mean, *M*, of the norms of the magnitude of Zn over all n cases is 5.8 and it was used to compute the ratio of testing RMSE over *M* as R_{AEC}, or 0.0067.

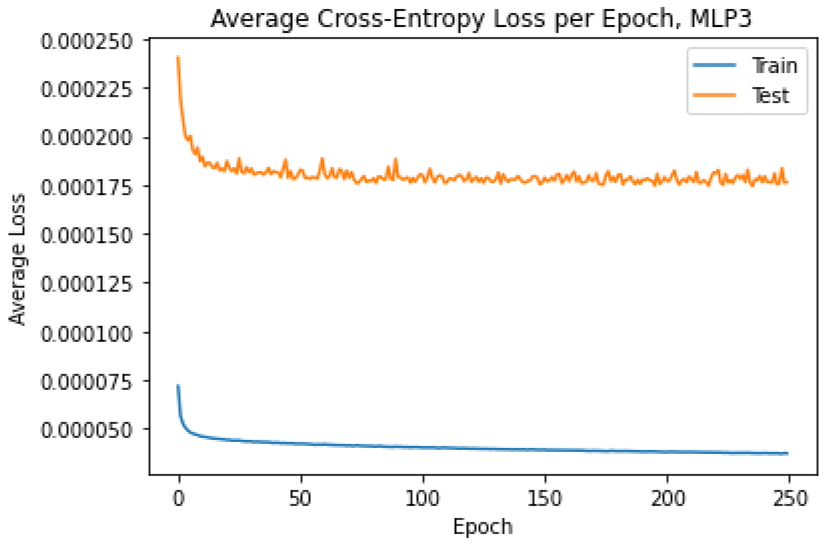
The testRMSE(m_{AEC}) represents the root mean squared error (RMSE) for a model's predictions on a test set, using the mean absolute error criterion (m_{AEC}) as the evaluation metric. The lower the testRMSE(m_{AEC}), the better the model's performance in predicting the test set.

On the other hand, M represents the mean of the norms ||Zn|| over all cases #n. The norms ||Zn|| measure the magnitude of a vector, and taking the mean over all cases #n provides an overall measure of the "size" of the vectors.The ratio R_{AEC} can be interpreted as the ratio of the model's performance on the test set, as measured by testRMSE(m_{AEC}), relative to the overall "size" of the vectors, as measured by M. A lower value of R_{AEC} suggests that the model's performance is relatively good compared to the "size" of the vectors.

In this case, we have R_{AEC} = 0.0067, indicating that the model's performance on the test set, as measured by testRMSE(m_{AEC}), is quite good relative to the overall "size" of the vectors, as measured by M.

We applied MLP_{12} to an original input vector Xn of dimension p, the activity of hidden layer H becomes vector Zn of dimension h= h_{high}, and the activity of layer K becomes a vector U_n of dimension dimK= h_{high}. We now use each U_n as a new input vector describing the case ”n”. These new input vectors are going to be used to train a new classifier, MLP_3, which has the test accuracy of 70.83% as the figure and confusion matrix regarding *test_data* below:

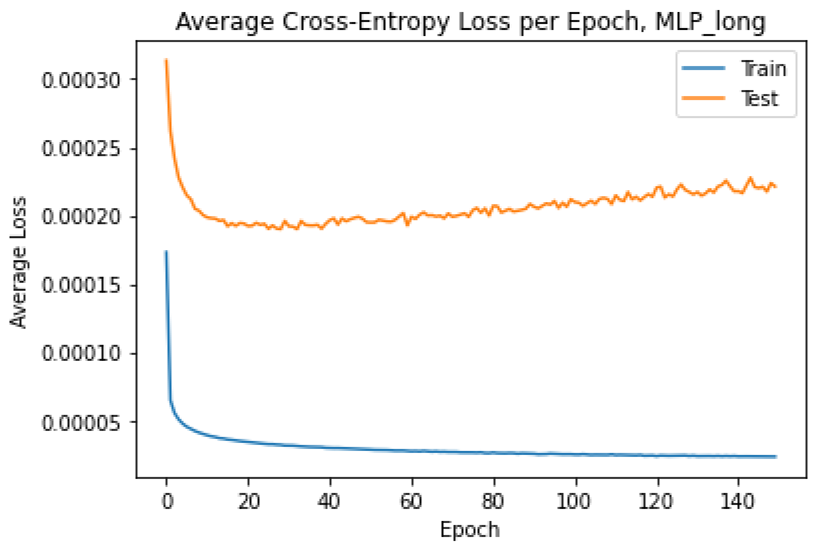
Ravik Chand
Lucas Smith
Huy Nguyen



	CL1	**CL2**	**CL3**	**CL4**	**CL5**	**CL6**
CL1	77.86	20.12	1.74	0.14	0	0.14
CL2	16.64	73.71	8.99	0.33	0.33	0
CL3	3.48	15.38	71.41	8.71	0.29	0.73
CL4	0.3	4.96	19.85	68.72	5.26	0.9
CL5	0.38	1.65	5.47	20.61	55.85	16.03
CL6	0	0	0.64	1.93	20.57	76.86

Now we build another model which is MLP_{LONG}. It is the concatenation of MLP_{12} and MLP_3* with classifier with 3 hidden layers H,K,G: L1 **->** H**->** K **->**G **->** OUT **->** softmax.Our MLP_{LONG} model has the test accuracy of 72.63% and has performance like below:

Ravik Chand
Lucas Smith
Huy Nguyen



The final result of our model are show as the table blow:

MLP	***m***	**TEST Acc. (%)**	**Compute Time(sec)**
MLP_INIT	175	66.37	37.15
MLP_LOW	225	61.69	34.95
MLP_PCA	160	61.93	35.90
MLP_HIGH	160	68.57	32.92
MLP3	18	70.48	24.30
MLP_LONG	18	70.45	29.88

The compute times shown are the average computation time per epoch, multiplied by *m*, the ideal stopping epoch. As we can see, MLP_{LONG} and MLP3 have almost the same accuracy, therefore they are our best performance models of every model we built. However, due to the additional layers of MLP_LONG, which increase computing time, MLP3 is selected as our best model overall.

MATH 6373 - Deep Learning and Neural Networks
Multilayer Perceptron Project Report

Ravik Chand
Lucas Smith
Huy Nguyen

Code Part

Q0:

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.metrics import log_loss
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import OneHotEncoder, StandardScaler
data = pd.read_csv("train.csv")

data = data.dropna()

quantiles = np.quantile(data["critical_temp"], np.arange(0, 1.1, 0.1))

e1 = data[data["critical_temp"] < 3.32]
e2 = data[(data["critical_temp"] > 3.32) & (data["critical_temp"] < 8.4)]
e3 = data[(data["critical_temp"] > 8.4) & (data["critical_temp"] < 20)]
e4 = data[(data["critical_temp"] > 20) & (data["critical_temp"] < 40.56667)]
e5 = data[(data["critical_temp"] > 40.56667) & (data["critical_temp"] < 79)]
e6 = data[data["critical_temp"] > 79]

column_names = pd.DataFrame(data.columns)

data.head()

# Add class labels, drop temperature
e1["label"] = "CL1"
e1 = e1.drop("critical_temp", axis=1)

e2["label"] = "CL2"
e2 = e2.drop("critical_temp", axis=1)

e3["label"] = "CL3"
e3 = e3.drop("critical_temp", axis=1)

e4["label"] = "CL4"
e4 = e4.drop("critical_temp", axis=1)

e5["label"] = "CL5"
e5 = e5.drop("critical_temp", axis=1)
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
e6["label"] = "CL6"
e6 = e6.drop("critical_temp", axis=1)

e1.head()

Q1:
#Q1: We dont need to use SMOTE and one-hot encoding
# Randomly select 80% of cases from each class to create the training set
train_e1, test_e1 = train_test_split(e1, test_size=0.2, random_state=42)
train_e2, test_e2 = train_test_split(e2, test_size=0.2, random_state=42)
train_e3, test_e3 = train_test_split(e3, test_size=0.2, random_state=42)
train_e4, test_e4 = train_test_split(e4, test_size=0.2, random_state=42)
train_e5, test_e5 = train_test_split(e5, test_size=0.2, random_state=42)
train_e6, test_e6 = train_test_split(e6, test_size=0.2, random_state=42)

# Concatenate the training and test sets for each class to create the final TRAIN and TEST sets
train_data = pd.concat([train_e1, train_e2, train_e3, train_e4, train_e5, train_e6])
test_data = pd.concat([test_e1, test_e2, test_e3, test_e4, test_e5, test_e6])

# One-hot encoding
# train data
one_hot_train = pd.get_dummies(train_data["label"])
train_data = train_data.drop("label", axis=1)
train_data = pd.concat([train_data, one_hot_train], axis=1)

# test data
one_hot_test = pd.get_dummies(test_data["label"])
test_data = test_data.drop("label", axis=1)
test_data = pd.concat([test_data, one_hot_test], axis=1)

train_data.head()

#Q2:
# Split the dataset into features and target variable
en_col = ["CL1", "CL2", "CL3", "CL4", "CL5", "CL6"]

x_train = train_data.drop(en_col, axis=1)
y_train = train_data[en_col]
x_test = test_data.drop(en_col, axis=1)
y_test = test_data[en_col]

# Standardize the features
scaler = StandardScaler()
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
x_train = pd.DataFrame(scaler.fit_transform(x_train))
x_test = pd.DataFrame(scaler.transform(x_test))

x_train.head()
from tensorflow.keras.initializers import RandomUniform
import time

# Validation Split
x_train1, x_valid, y_train1, y_valid = train_test_split(x_train, y_train, test_size=0.1)

# Define the model
model = tf.keras.Sequential([
   tf.keras.layers.Dense(41, activation='relu', input_shape=(x_train.shape[1],),
kernel_initializer=RandomUniform),
   tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
start_time = time.time()

history = model.fit(x_train1, y_train1, batch_size=152, epochs=175, verbose=0,
validation_data=(x_valid,y_valid))

end_time = time.time()

time_per_epoch = (end_time - start_time) / len(history.history['loss'])

# Calculate average cross-entropy loss for train and test sets
train_preds = model.predict(x_train)
train_avCRE = log_loss(y_train, train_preds) / len(x_train)

test_preds = model.predict(x_test)
test_avCRE = log_loss(y_test, test_preds) / len(x_test)

# Extract the training and validation losses for each epoch
train_losses_per_epoch = history.history['loss']
test_losses_per_epoch = history.history['val_loss']

# Calculate the average cross-entropy loss per epoch for train and test sets
train_avCRE_per_epoch = [loss / len(x_train) for loss in train_losses_per_epoch]
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
test_avCRE_per_epoch = [loss / len(x_test) for loss in test_losses_per_epoch]

print("Average Cross Entropy Loss for Train Set:", train_avCRE)
print("Average Cross Entropy Loss for Test Set:", test_avCRE)

print("Average Cross Entropy Loss per Epoch for Train Set:", train_avCRE_per_epoch)
print("Average Cross Entropy Loss per Epoch for Test Set:", test_avCRE_per_epoch)

print("Computing time per epoch: ", time_per_epoch)

# 150 epochs was originally used, but the accuracy of the training set was still
# continuing to decrease, so I bumped it up to 200
# Computation time for 150 epochs: ~3.5 min
# Computation time for 200 epochs: ~4.5 min
# Computation time for 250 epochs: ~5.5 min
# We will use 150 epochs going forward. Test accuracy levels off at around this point.
# 200 and 250 epochs were computed to make sure no accuracy gains were left on the table.

import matplotlib.pyplot as plt

# Plot the loss per epoch
plt.plot(train_losses_per_epoch, label='Train')
plt.plot(test_losses_per_epoch, label='Test')
plt.title('Cross-Entropy Loss per Epoch, h=41')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)

print("Test Accuracy: {:.2f}%".format(test_accuracy * 100))
round(x_train1.shape[0]/100, 0)
# Create and print the confusion matrix.

from sklearn.metrics import confusion_matrix

test_pred_labels = np.argmax(test_preds, axis=1)
y_test_array = y_test.to_numpy()
y_test_labels = np.argmax(y_test_array, axis=1)

# Compute the confusion matrix
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
conf_matrix = confusion_matrix(test_pred_labels,y_test_labels)

# Normalize the confusion matrix
normalized_conf_matrix = conf_matrix.astype('float') / conf_matrix.sum(axis=1)[:, np.newaxis]

# Convert to percentage
percentage_conf_matrix = normalized_conf_matrix * 100

print("Confusion Matrix (Percentage):")
print("Confusion Matrix (Percentage):")
for row in percentage_conf_matrix:
    formatted_row = ["{:.2f}".format(x) for x in row]
    print(" ".join(formatted_row))
```

Q3:

```
from sklearn.decomposition import PCA

# Create a PCA object with 90% variance explained
pca = PCA(n_components=0.9)

# Fit the PCA model on the training data
pca.fit(x_train)

# Transform the training data into the new feature space
x_train_pca = pca.transform(x_train)
x_test_pca = pca.transform(x_test)

x_train_pca = pd.DataFrame(x_train_pca)
x_test_pca = pd.DataFrame(x_test_pca)

# Print the shape of the transformed data
print("Shape of transformed data:", x_train_pca.shape)

# Create a PCA object
pca = PCA()

# Fit the PCA model on the training data
pca.fit(x_train)

# Compute the cumulative sum of explained variance
var_exp = np.cumsum(pca.explained_variance_ratio_)

# Create a bar plot of explained variance vs. number of components
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
plt.figure(figsize=(8, 6))
plt.bar(range(1, len(var_exp) + 1), var_exp, align='center', label='Individual explained variance')
plt.step(range(1, len(var_exp) + 1), var_exp, where='mid', label='Cumulative explained
variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.legend(loc='best')
plt.tight_layout()
plt.show()

# Validation Split
x_train2, x_valid, y_train2, y_valid = train_test_split(x_train, y_train, test_size=0.1)

# Define the model
model2 = tf.keras.Sequential([
   tf.keras.layers.Dense(12, activation='relu', input_shape=(x_train.shape[1],),
kernel_initializer=RandomUniform),
   tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

# Compile the model
model2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
start_time = time.time()

history2 = model2.fit(x_train2, y_train2, batch_size=152, epochs=225, verbose=0,
validation_data=(x_valid,y_valid))

end_time = time.time()

time_per_epoch2 = (end_time - start_time) / len(history2.history['loss'])

# Calculate average cross-entropy loss for train and test sets
train_preds2 = model2.predict(x_train)
train_avCRE2 = log_loss(y_train, train_preds2) / len(x_train)

test_preds2 = model2.predict(x_test)
test_avCRE2 = log_loss(y_test, test_preds2) / len(x_test)

# Extract the training and validation losses for each epoch
train_losses_per_epoch2 = history2.history['loss']
test_losses_per_epoch2 = history2.history['val_loss']
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
# Calculate the average cross-entropy loss per epoch for train and test sets
train_avCRE_per_epoch2 = [loss / len(x_train) for loss in train_losses_per_epoch2]
test_avCRE_per_epoch2 = [loss / len(x_test) for loss in test_losses_per_epoch2]

print("Average Cross Entropy Loss for Train Set:", train_avCRE2)
print("Average Cross Entropy Loss for Test Set:", test_avCRE2)

print("Average Cross Entropy Loss per Epoch for Train Set:", train_avCRE_per_epoch2)
print("Average Cross Entropy Loss per Epoch for Test Set:", test_avCRE_per_epoch2)

print("Computing time per epoch: ", time_per_epoch2)

# Plot the loss per epoch
plt.plot(train_losses_per_epoch2, label='Train')
plt.plot(test_losses_per_epoch2, label='Test')
plt.title('Cross-Entropy Loss per Epoch, h_low=12')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Evaluate the model on the test set
test_loss2, test_accuracy2  = model2.evaluate(x_test, y_test, verbose=0)

print("Test Accuracy: {:.2f}%".format(test_accuracy2 * 100))

# Need to work out why this model has over 100% accuracy. Will come back to it.

# There is a higher loss than the first model, but not by much.

test_pred_labels2 = np.argmax(test_preds2, axis=1)

# Compute the confusion matrix
conf_matrix2 = confusion_matrix(test_pred_labels2,y_test_labels)

# Normalize the confusion matrix
normalized_conf_matrix2 = conf_matrix2.astype('float') / conf_matrix2.sum(axis=1)[:,
np.newaxis]

# Convert to percentage
percentage_conf_matrix2 = normalized_conf_matrix2 * 100
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
print("Confusion Matrix (Percentage):")
print("Confusion Matrix (Percentage):")
for row in percentage_conf_matrix2:
    formatted_row = ["{:.2f}".format(x) for x in row]
    print(" ".join(formatted_row))
```

Q4:

```
from sklearn.decomposition import PCA

# Define the variance threshold
var_threshold = 0.9

# Loop through each dataframe and perform PCA on numeric columns
for i, df in enumerate([e1, e2, e3, e4, e5, e6]):
    numeric_df = df.select_dtypes(include=['float64', 'int64'])
    pca = PCA(n_components=var_threshold)
    pca.fit(numeric_df)
    n_components = pca.n_components_
    print(f"Number of features that capture 90% of the variance for e{i+1}: {n_components}")

# Our variance is highly concentrated to only a few features for every class.
# h1 + h2 ... = 13, which is the same dimension used for h_low.
# Going to use h=p where p is the total number of features for h_high

# Define the variance threshold
var_threshold = 0.9

# Define the number of principal components to try
max_components = 10

# Loop through each dataframe and perform PCA on numeric columns
for i, df in enumerate([e1, e2, e3, e4, e5, e6]):
    numeric_df = df.select_dtypes(include=['float64', 'int64'])
    pca = PCA(n_components=max_components)
    pca.fit(numeric_df)
    var_ratio = pca.explained_variance_ratio_
    cum_var_ratio = np.cumsum(var_ratio)
    n_components = np.argmax(cum_var_ratio >= var_threshold) + 1
    plt.plot(np.arange(1, max_components+1), cum_var_ratio, label=f'e{i+1}')

plt.xlabel('Number of principal components')
plt.ylabel('Percentage of explained variance')
plt.axhline(y=var_threshold, color='black', linestyle='--', label=f'{var_threshold:.0%} threshold')
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
plt.legend()
plt.show()

# Validation Split
x_train3, x_valid, y_train3, y_valid = train_test_split(x_train, y_train, test_size=0.1)

# Define the model
model3 = tf.keras.Sequential([
   tf.keras.layers.Dense(13, activation='relu', input_shape=(x_train.shape[1],),
kernel_initializer=RandomUniform),
   tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

# Compile the model
model3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
start_time = time.time()

history3 = model3.fit(x_train3, y_train3, batch_size=152, epochs=160, verbose=0,
validation_data=(x_valid,y_valid))

end_time = time.time()

time_per_epoch3 = (end_time - start_time) / len(history3.history['loss'])

# Calculate average cross-entropy loss for train and test sets
train_preds3 = model3.predict(x_train)
train_avCRE3 = log_loss(y_train, train_preds3) / len(x_train)

test_preds3 = model3.predict(x_test)
test_avCRE3 = log_loss(y_test, test_preds3) / len(x_test)

# Extract the training and validation losses for each epoch
train_losses_per_epoch3 = history3.history['loss']
test_losses_per_epoch3 = history3.history['val_loss']

# Calculate the average cross-entropy loss per epoch for train and test sets
train_avCRE_per_epoch3 = [loss / len(x_train) for loss in train_losses_per_epoch3]
test_avCRE_per_epoch3 = [loss / len(x_test) for loss in test_losses_per_epoch3]

print("Average Cross Entropy Loss for Train Set:", train_avCRE3)
print("Average Cross Entropy Loss for Test Set:", test_avCRE3)
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
print("Average Cross Entropy Loss per Epoch for Train Set:", train_avCRE_per_epoch3)
print("Average Cross Entropy Loss per Epoch for Test Set:", test_avCRE_per_epoch3)

print("Computing time per epoch: ", time_per_epoch3)

# Plot the loss per epoch
plt.plot(train_losses_per_epoch3, label='Train')
plt.plot(test_losses_per_epoch3, label='Test')
plt.title('Cross-Entropy Loss per Epoch,  h_3=13')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Evaluate the model on the test set
test_loss3, test_accuracy3  = model3.evaluate(x_test, y_test, verbose=0)

print("Test Accuracy: {:.2f}%".format(test_accuracy3 * 100))

test_pred_labels3 = np.argmax(test_preds3, axis=1)

# Compute the confusion matrix
conf_matrix3 = confusion_matrix(test_pred_labels3,y_test_labels)

# Normalize the confusion matrix
normalized_conf_matrix3 = conf_matrix3.astype('float') / conf_matrix3.sum(axis=1)[:,
np.newaxis]

# Convert to percentage
percentage_conf_matrix3 = normalized_conf_matrix3 * 100

print("Confusion Matrix (Percentage):")
print("Confusion Matrix (Percentage):")
for row in percentage_conf_matrix3:
    formatted_row = ["{:.2f}".format(x) for x in row]
    print(" ".join(formatted_row))

# Validation Split
x_train4, x_valid, y_train4, y_valid = train_test_split(x_train, y_train, test_size=0.1)

# Define the model
model4 = tf.keras.Sequential([
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
    tf.keras.layers.Dense(81, activation='relu', input_shape=(x_train.shape[1],),
kernel_initializer=RandomUniform),
    tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

# Compile the model
model4.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
start_time = time.time()

history4 = model4.fit(x_train4, y_train4, batch_size=152, epochs=160, verbose=0,
validation_data=(x_valid,y_valid))

end_time = time.time()

time_per_epoch4 = (end_time - start_time) / len(history4.history['loss'])

# Calculate average cross-entropy loss for train and test sets
train_preds4 = model4.predict(x_train)
train_avCRE4 = log_loss(y_train, train_preds4) / len(x_train)

test_preds4 = model4.predict(x_test)
test_avCRE4 = log_loss(y_test, test_preds4) / len(x_test)

# Extract the training and validation losses for each epoch
train_losses_per_epoch4 = history4.history['loss']
test_losses_per_epoch4 = history4.history['val_loss']

# Calculate the average cross-entropy loss per epoch for train and test sets
train_avCRE_per_epoch4 = [loss / len(x_train) for loss in train_losses_per_epoch4]
test_avCRE_per_epoch4 = [loss / len(x_test) for loss in test_losses_per_epoch4]

print("Average Cross Entropy Loss for Train Set:", train_avCRE4)
print("Average Cross Entropy Loss for Test Set:", test_avCRE4)

print("Average Cross Entropy Loss per Epoch for Train Set:", train_avCRE_per_epoch4)
print("Average Cross Entropy Loss per Epoch for Test Set:", test_avCRE_per_epoch4)

print("Computing time per epoch: ", time_per_epoch4)

# Plot the loss per epoch
plt.plot(train_losses_per_epoch4, label='Train')
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
plt.plot(test_losses_per_epoch4, label='Test')
plt.title('Cross-Entropy Loss per Epoch,  h_high=81')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Evaluate the model on the test set
test_loss4, test_accuracy4  = model4.evaluate(x_test, y_test, verbose=0)

print("Test Accuracy: {:.2f}%".format(test_accuracy4 * 100))

test_pred_labels4 = np.argmax(test_preds4, axis=1)

# Compute the confusion matrix
conf_matrix4 = confusion_matrix(test_pred_labels4,y_test_labels)

# Normalize the confusion matrix
normalized_conf_matrix4 = conf_matrix4.astype('float') / conf_matrix4.sum(axis=1)[:,
np.newaxis]

# Convert to percentage
percentage_conf_matrix4 = normalized_conf_matrix4 * 100

print("Confusion Matrix (Percentage):")
print("Confusion Matrix (Percentage):")
for row in percentage_conf_matrix4:
    formatted_row = ["{:.2f}".format(x) for x in row]
    print(" ".join(formatted_row))
```

Q5:

```
# Define the model, this time with seperate objects for each layer
input_layer = tf.keras.Input(shape=(x_train.shape[1],))
hidden_layer = tf.keras.layers.Dense(81, activation='sigmoid',
kernel_initializer=RandomUniform)(input_layer)
output_layer = tf.keras.layers.Dense(y_train.shape[1], activation='softmax')(hidden_layer)

mlp_high = tf.keras.Model(inputs=input_layer, outputs=output_layer)

# Compile the model
mlp_high.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
history = mlp_high.fit(x_train, y_train, batch_size=42, epochs=150, verbose=0,
validation_data=(x_test,y_test))

# Create new model to study the hidden layer
hidden_layer_model = tf.keras.Model(inputs=mlp_high.input, outputs=mlp_high.layers[1].output)

# Compute Zn for each case in x_train
Zn_train = hidden_layer_model.predict(x_train)
Zn_test = hidden_layer_model.predict(x_test)

latent_dim = 81
input_dim = 81
sparsity_target = 0.1
sparsity_weight = 1e-3

class Autoencoder(tf.keras.Model):
    def __init__(self, latent_dim, input_dim):
        super(Autoencoder, self).__init__()

        self.latent_dim = latent_dim
        self.encoder = tf.keras.Sequential([
            tf.keras.layers.Input(shape=(input_dim,)),
            tf.keras.layers.Dense(latent_dim, activation='sigmoid'),
        ])

        self.decoder = tf.keras.Sequential([
            tf.keras.layers.Dense(input_dim, activation='sigmoid'),
        ])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

autoencoder = Autoencoder(latent_dim, input_dim)

# Define RMSE as a custom metric
def rmse(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))

# Define the KL divergence sparsity penalty
def kl_divergence(p, p_hat):
    return p * tf.math.log(p / p_hat) + (1 - p) * tf.math.log((1 - p) / (1 - p_hat))
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
# Define the sparse loss function
def sparse_loss(y_true, y_pred):
    mse_loss = tf.reduce_mean(tf.square(y_pred - y_true))
    hidden_layer = autoencoder.encoder(y_true)
    p_hat = tf.reduce_mean(hidden_layer, axis=0)
    sparsity_loss = tf.reduce_sum(kl_divergence(sparsity_target, p_hat))
    total_loss = mse_loss + sparsity_weight * sparsity_loss
    return total_loss

autoencoder.compile(optimizer='adam', loss=sparse_loss, metrics=[rmse])

history_aec = autoencoder.fit(Zn_train, Zn_train,
                              epochs=75,
                              shuffle=True,
                              validation_data=(Zn_test, Zn_test),
                              verbose=1)

train_rmse = history_aec.history['rmse']
val_rmse = history_aec.history['val_rmse']
epochs = range(1, len(train_rmse) + 1)

plt.plot(epochs, train_rmse, 'b', label='Training RMSE')
plt.plot(epochs, val_rmse, 'r', label='Test RMSE')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.legend()
plt.title('RMSE vs. Epochs')

plt.show()

# Copy and paste model3 from other file

# Define the model
model3 = tf.keras.Sequential([
    tf.keras.layers.Dense(81, activation='relu', input_shape=(x_train.shape[1],),
kernel_initializer=RandomUniform),
    tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

# Compile the model
model3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
# Train the model
history3 = model3.fit(x_train, y_train, batch_size=42, epochs=150, verbose=0,
validation_data=(x_test,y_test))

# Extract weights and biases from the first dense layer of model3 and the encoder
model3_dense1_weights, model3_dense1_biases = model3.layers[0].get_weights()

encoder_weights, encoder_biases = autoencoder.encoder.layers[0].get_weights()

# Verify that the weights and biases have the same dimensions
print("Model3 Weights Size: ", model3_dense1_weights.shape)
print("Model3 Biases Size: ", model3_dense1_biases.shape)
print("Encoder Weights Size: ", encoder_weights.shape)
print("Encoder Biases Size: ", encoder_biases.shape)

Zn_combined = np.concatenate((Zn_train, Zn_test), axis=0)

# Define the modified MLP_12 model
MLP_12 = tf.keras.Sequential([
    model3.layers[0],
    autoencoder.encoder,
    tf.keras.layers.Dense(81, activation='linear')  # Change the output dimension to 82
])

# Compile the modified model
MLP_12.compile(optimizer='adam', loss='mse')

# Train the modified model
history_MLP_12 = MLP_12.fit(x_train, Zn_train,
                            batch_size=42,
                            epochs=150,
                            verbose=0,
                            validation_data=(x_test, Zn_test))

# Combine x_train and x_test
x_combined = np.concatenate((x_train, x_test), axis=0)
y_combined = np.concatenate((y_train, y_test), axis=0)

# Predict the output using the modified MLP_12 model
U_n = MLP_12.predict(x_combined)

# Split the U_n dataset into training and test sets (80/20)
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
U_n_train, U_n_test, y_train_new, y_test_new = train_test_split(U_n, y_combined,
test_size=0.2, random_state=42)
U_n.shape

# Define the MLP3 model
MLP3 = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(U_n_train.shape[1],)),
    tf.keras.layers.Dense(40, activation='relu'),
    tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

# Compile the MLP3 model
MLP3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the MLP3 model
history_MLP3 = MLP3.fit(U_n_train, y_train_new,
                        batch_size=42,
                        epochs=250,
                        verbose=0,
                        validation_data=(U_n_test, y_test_new))

train_preds =  MLP3.predict(U_n_train)
train_avCRE = log_loss(y_train_new, train_preds) / len(U_n_train)

test_preds = MLP3.predict(U_n_test)
test_avCRE = log_loss(y_test_new, test_preds) / len(U_n_test)

# Extract the training and validation losses for each epoch
train_losses_per_epoch = history_MLP3.history['loss']
test_losses_per_epoch = history_MLP3.history['val_loss']

# Calculate the average cross-entropy loss per epoch for train and test sets
train_avCRE_per_epoch = [loss / len(U_n_train) for loss in train_losses_per_epoch]
test_avCRE_per_epoch = [loss / len(U_n_test) for loss in test_losses_per_epoch]

print("Average Cross Entropy Loss for Train Set:", train_avCRE)
print("Average Cross Entropy Loss for Test Set:", test_avCRE)

print("Average Cross Entropy Loss per Epoch for Train Set:", train_avCRE_per_epoch)
print("Average Cross Entropy Loss per Epoch for Test Set:", test_avCRE_per_epoch)

plt.plot(train_avCRE_per_epoch, label='Train')
plt.plot(test_avCRE_per_epoch, label='Test')
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
plt.title('Average Cross-Entropy Loss per Epoch, MLP3')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.legend()
plt.show()

# Evaluate the MLP3 model on the U_n_test set
test_loss_MLP3, test_accuracy_MLP3 = MLP3.evaluate(U_n_test, y_test_new, verbose=0)

print("Test Accuracy: {:.2f}%".format(test_accuracy_MLP3 * 100))

# Create and print the confusion matrix.
test_pred_labels = np.argmax(test_preds, axis=1)
y_test_array = y_test_new
y_test_labels = np.argmax(y_test_array, axis=1)

# Compute the confusion matrix
conf_matrix = confusion_matrix(test_pred_labels, y_test_labels)

# Normalize the confusion matrix
normalized_conf_matrix = conf_matrix.astype('float') / conf_matrix.sum(axis=1)[:, np.newaxis]

# Convert to percentage
percentage_conf_matrix = normalized_conf_matrix * 100

print("Confusion Matrix (Percentage):")
print("Confusion Matrix (Percentage):")
for row in percentage_conf_matrix:
   formatted_row = ["{:.2f}".format(x) for x in row]
   print(" ".join(formatted_row))
MLP_long = tf.keras.Sequential([
   MLP_12.layers[0],  # First hidden layer from MLP_12
   MLP_12.layers[1],  # Second hidden layer (output layer of MLP_12)
   MLP3.layers[0],  #
   MLP3.layers[1],  # Third hidden layer from MLP3
])


# Compile the MLP_long model
MLP_long.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


# Train the MLP_long model
```

Ravik Chand
Lucas Smith
Huy Nguyen

```
history_MLP_long = MLP_long.fit(x_train, y_train,
                        batch_size=42,
                        epochs=50,
                        verbose=0,
                        validation_data=(x_test, y_test))

train_preds_long =  MLP_long.predict(x_train)
train_avCRE_long = log_loss(y_train, train_preds_long) / len(x_train)

test_preds_long = MLP_long.predict(x_test)
test_avCRE_long = log_loss(y_test, test_preds_long) / len(x_test)

# Extract the training and validation losses for each epoch
train_losses_per_epoch_long = history_MLP_long.history['loss']
test_losses_per_epoch_long = history_MLP_long.history['val_loss']

# Calculate the average cross-entropy loss per epoch for train and test sets
train_avCRE_per_epoch_long = [loss / len(x_train) for loss in train_losses_per_epoch_long]
test_avCRE_per_epoch_long = [loss / len(x_test) for loss in test_losses_per_epoch_long]

print("Average Cross Entropy Loss for Train Set:", train_avCRE_long)
print("Average Cross Entropy Loss for Test Set:", test_avCRE_long)

print("Average Cross Entropy Loss per Epoch for Train Set:", train_avCRE_per_epoch_long)
print("Average Cross Entropy Loss per Epoch for Test Set:", test_avCRE_per_epoch_long)

plt.plot(train_avCRE_per_epoch_long, label='Train')
plt.plot(test_avCRE_per_epoch_long, label='Test')
plt.title('Average Cross-Entropy Loss per Epoch, MLP_long')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.legend()
plt.show()

# Evaluate the MLP3 model on the U_n_test set
test_loss_long, test_accuracy_long = MLP_long.evaluate(x_test, y_test, verbose=0)

print("Test Accuracy: {:.2f}%".format(test_accuracy_long * 100))
```