Processing Images and Data with Deep Neural Networks (DNN)

Joseph Tresise
School of Computing and Communications
Lancaster University

***Abstract*—This report focuses on two completion of two related tasks. The first task processes climate data and clusters the results using two different clustering algorithms (K-Means and DBSCAN). The second task involves the processing of images from two datasets utilsing a pretrained ResNet-18 Deep Neural Network and displays the results.**

***Index Terms*—Deep Neural Networks, ResNet-18, PCA, K-Means, DBSCAN**

I. Introduction

This report focuses on the application and performance of data analysis in two separate tasks. Both tasks used **Python Version *3.12.4***, and ran within their own dedicated **Jupyter Notebooks**. A full list of the libraries used for these tasks can be found within the ***Acknowledgment*** section.

The first task involved the preprocessing and subsequent clustering of a climate dataset. This set of climate data is entitled *'ClimateDataBasel.csv'* and will subsequently be referred to as the ***Basel Climate Data***. This data contained **1,738** 18-dimensional records of data. For this task, the objective was to first preprocess the data so that the data would be clean and consistent. This consistency then made clustering significantly easily and produced better results.

The second task focuses on the use of a pretrained **Deep Neural Network (DNN)** to be used for image processing. In short, DNNs transform complex multidimensional data, for example *images* into lower-dimension, information-dense representations. These representations are known as feature vectors, and these vectors form a *latent space* in which objects that are similar will be mapped near to each other whilst dissimilar objects are placed further apart. The objective of this of task was to use a DNN to extract feature vectors and then further analyze them utilizing visualization, clustering, and classification. This will be explored in ***Image Processing with Deep Neural Networks***.

Basel Climate Data

II. Preprocessing

Pre-Processing is the first step within data mining, and the core of is this step revolves around the idea of the dataset needing to be prepared and '*cleaned*' before it can be properly analysed, or used to build models. With raw data, it is often found that the collected data is either incomplete, noisy, or just inconsistent. Typically it is a combination of all of these issues, and as such Pre-Processing helps to mold this data in a '*clean*' and structured format. This invariably helps the later processing of the data whilst also improving the quality and accuracy of any analysis performed.

After reading the file into Python via Jupyter, and displaying the first ten entries of Basel Climate Data. I took the initial data and generated box-plots with the aim of this informing outlier removal.

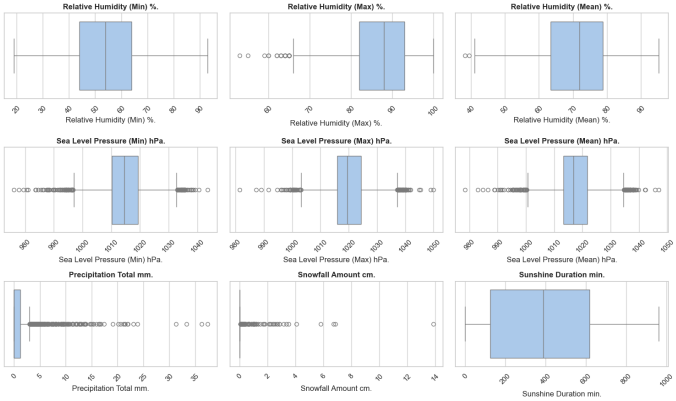

Fig. 1. Subset of Box-plots from Initial Data.

As seen in Figure 1 above, there were numerous outliers present within the different categories of data. To combat this, I decided to implement the Inter-Quartile-Range (IQR) method for outlier removal. IQR is used a measure of statistical dispersion, and it shows the range within where the *50%* between *25%* and *75%* lies. Both *25%* and *75%* are referred to as the first and third quantile respectively. The formula to calculate the Inter-Quartile-Range is:

$$\boldsymbol{IQR} = Q_3 - Q_1 \tag{1}$$

To use IQR in outlier removal, it is generally accepted that observations within the dataset are classified as outliers when they *'lie outside'* 1.5 IQR below Q1 and 1.5 IQR above Q3. Therefore it can be said that:

$$\textbf{\textit{Outliers}} = Q_1 - 1.5 \times \text{IQR} \tag{2}$$

OR

$$\textbf{\textit{Outliers}} = Q_3 + 1.5 \times \text{IQR} \tag{3}$$

I choose to use IQR for numerous reasons. The main advantage of using the IQR method is that identifies based on percentiles, therefore making it less sensitive to extreme values which in turn makes it more robust when dealing with a skewed data distribution. The IQR method is also easy to implement, making it very effective when dealing with the data.

Within my code, I created an IQR function which removed outliers at both the .25 (*Q1*) and .75 quantiles (*Q3*) within the data set. This was calculated using Figures *1*, *2*, and *3*. Both of these functions can be found within the appendix under **Appendix A** and **Appendix B**. The IQR function returned three parameters, these were the original data shape, the data shape after removal of the outliers, and the number of outliers. The statistics are listed below:

Original Data Shape: (1763, 18)
After Outlier Removal: (1179, 18)
Outliers: (584, 18)

With these figures I then plotted a scatterplot of the outliers v.s. the non-outliers to show a general trend and the differences between both groups.

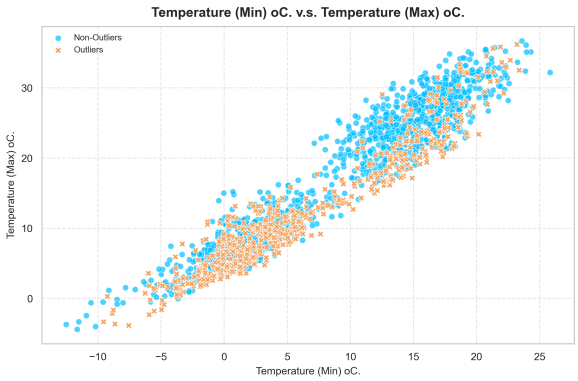

Fig. 2. Scatterplot of Temperature (Min) v.s Temperature (Max) showing both Outliers and Non-Outliers.

Here, Figure 3 shows a scatterplot from the first two columns of the dataset. However, it should be noted that this example is only displayed because the function produced this plot first, and it would be entirely possible to plot a scatterplot for any of the dimensions of the dataset, albeit making sure they confine to the rules of a scatterplot itself.

To deal with missing values and *NaNs* within the dataset I utilised two functions, *isna.()* and *fillna('NA')*, from the *Pandas* Python library. These functions find any missing values in a *Pandas DataFrame*and then replaces them value of **NA**. These functions were used just to catch any potential missing values that could have been present. The choice to use it after the IQR method was purely of my own volition but it could be used before and the outcome would be the same.

The final preprocessing step was to standardise and then normalise the data. Standardisation ensures that any features have a mean of **zero** and a standard deviation of **one**. Mathematically, the formula for standardisation is:

$$z = \frac{(X - \mu)}{\sigma} \tag{4}$$

X = Original (*raw*) score, μ = Mean, and σ = Standard Deviation.

For this project, I used the *StandardScaler* from the *sklearn.preprocessing* Python library. This utilises the above formula 4 and fits the standardisation scaler to the target dataset and then subsequently transforms it. The use of this standardisation scaler is critical for **Principal Component Analysis (PCA)** which assumes that the target data is of a Gaussian (Normal) distribution and that each feature is evenly weighted. Normalisation involves adjusting values of data to a scale where the values are in a pre-defined range. This is in the aim of making sure that every feature within a dataset does not dominate over another. Normalisation is use for avoiding bias within the K-means clustering algorithm. There are two types of Normalisation, these are **Min-Max** and **Z-Score**. Here, I utilised Min-Max as it scales data into the specified ranges of [0, 1] and [-1, 1]. The formula for this type of Normalisation is:

$$Xnorm = \frac{X - Xmin}{Xmax - Xmin} \tag{5}$$

$Xnorm$ = Normalised Value, $Xmin$ = Minimum Value of Feature, $Xmax$ = Maximum Value of Feature

Functions **C** and **D** are built on formula 5, and normalize data to [0,1] and [-1, 1] respectively. Shown below is a scatterplot of the normalised Basel Climate Data.

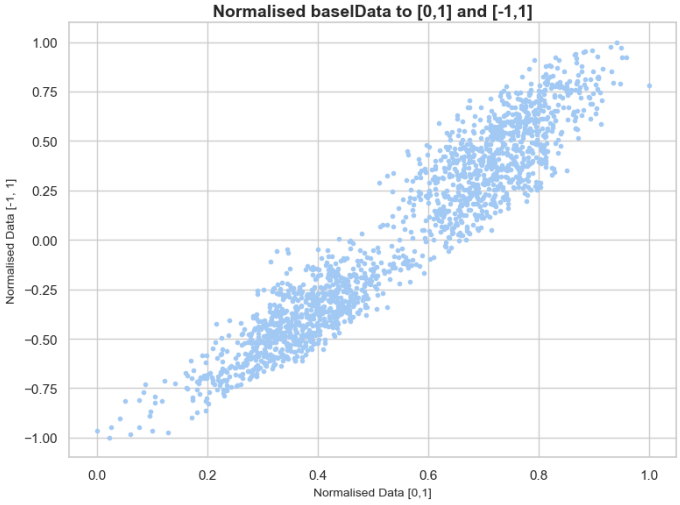

Fig. 3. Normalised Basel Data to [0,1] and [-1,1].

III. Clustering

Throughout Section II, the Basel Climate Dataset has been pre-processed. The first step of Clustering is to perform PCA. PCA reduces the number of variables, referred to as dimensions, into a smaller data set whilst extracting the most important features of the dataset with the aim of analysing variance between specific *'principal components'*. For the data in this project, I first centralised the data to subtract the mean of each feature column this helps for a later calcuation as part of the PCA process. At the core of PCA, is the calcuation and usage of a Correlation Matrix

$$\Sigma = \begin{bmatrix} \text{cov}(X_1, X_1) & \text{cov}(X_1, X_2) & \text{cov}(X_1, X_3) \\ \text{cov}(X_2, X_1) & \text{cov}(X_2, X_2) & \text{cov}(X_2, X_3) \\ \text{cov}(X_3, X_1) & \text{cov}(X_3, X_2) & \text{cov}(X_3, X_3) \end{bmatrix}$$

3-Dimensional Correlation Matrix

$\text{cov}(X_i, X_j)$ = Covariance between Variables X_i and X_j.

Where:

$$\text{cov}(X_i, X_j) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})$$

X_i, Y_i are the individual sample points of X and Y,$\bar{X}, \bar{Y}$ are the sample means of X and Y, n is the number of data points in the sample.

To calculate the correlation matrix within my work I utilised the *np.corrcoef* function from *Pandas*. This function takes the centralised data and computes the correlation matrix shown in Formula **??**. Once this matrix was computed, I then fitted the PCA to centralised data using the *PCA* function which is part of the *sklearn.preprocessing* library. This PCA function reduces the data to 18 principal components and fits it to the dataset. These principal components, are also known as **coefficients**, are eigenvectors of the correlation matrix.

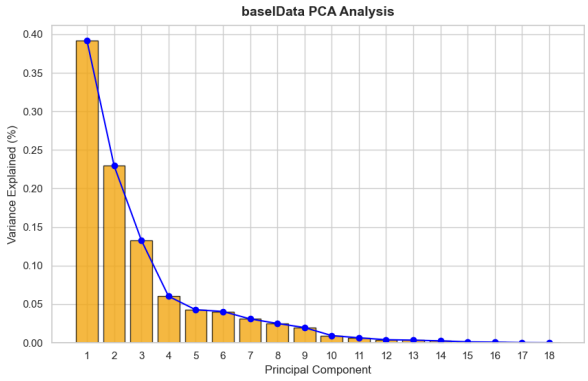

Fig. 4. Basel Data PCA Analysis.

Here, Figure 4 shows the results of the explained variance of the PCA Analysis. There is a clear decrease in explained variance from Principal Component One, which is approximately 0.39 (39%) of the total variance, to Principal Component 18, where the total variance appears to be less than 0.01 (1%).

For this project, I used two separate methods of Clustering, these are K-Means and DBSCAN Clustering. K-Means Clustering groups data points into a set of clusters. These clusters are based on similarity, at the center of each cluster is a ***centroid*** which represents the overall average for the cluster. Within, K-Means Clustering there is a method called the **Elbow Method** which selects the optimal number of k-Clusters. The Elbow Method plots the sum of squared distances from each point to the center (**Inertia/*Within-Cluster Sum of Squares (WCSS)***) for different values of K and the looking at the resultant to visually identify the *'elbow'* where the WCSS decreases at a rate slower than the rate of the increasing K.

$$\text{WCSS}(K) = \sum_{i=1}^{n} \sum_{k=1}^{K} \|x_i - c_k\|^2 \cdot \mathbb{I}_{(k)}(x_i) \tag{7}$$

Where:

- x_i = data point.
- c_k = centroid of the k-th cluster.
- $\|x_i - c_k\|$ = Euclidean distance between data point x_i and centroid c_k.
- $\mathbb{I}_{(k)}(x_i)$ = indicator function that is 1 if x_i belongs to cluster k, and 0 otherwise.
- K = number of clusters.

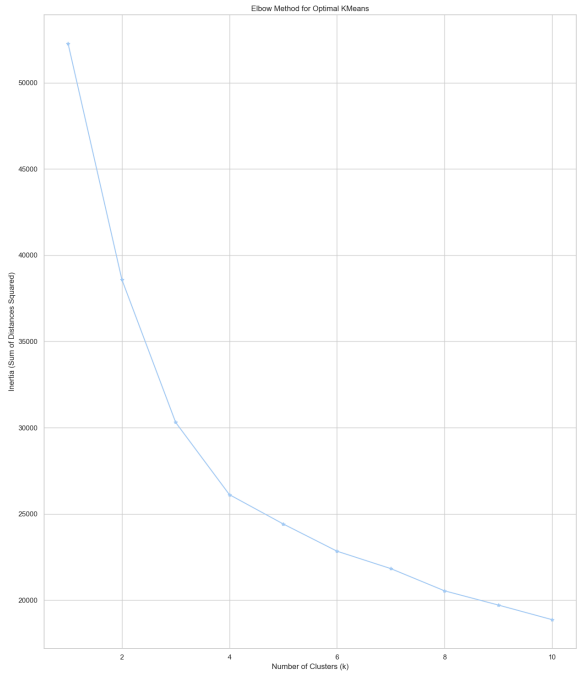

Fig. 5. Elbow Method for Optimal KMeans for Basel Data.

The above Figure 9 demonstrates the Elbow Method for the Basel Climate Data. If given more time and experimentation there is the potential that the optimal point could have been more accurately computed which could have improved the K-Means Clustering in comparison to the K-Means Clustering performed in this report. The Python library *scipy* was used to compute the K-Means clustering. Below is the outcome of this clustering method.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) focuses on distances between the nearest points expands the ideas of clusters within the K-Means Clustering algorithm. Within

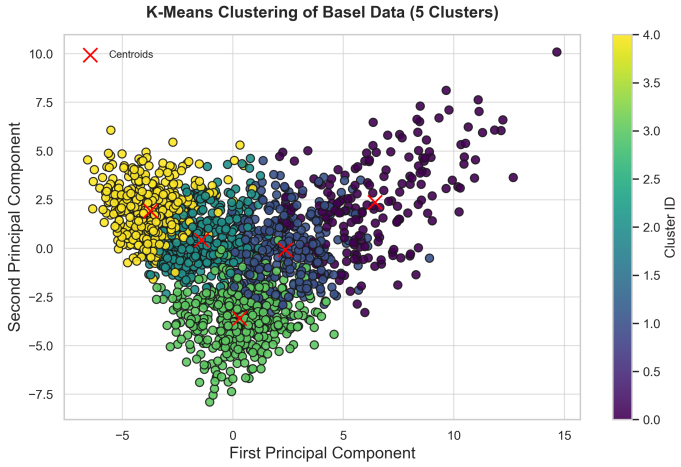

Fig. 6. K-Means Clustering of Basel Data.

DBSCAN, each point of a cluster must have a neighbourhood of a given radius which contains a specified minimum number of points. K-Means Clustering is works well to find convex clusters but they are greatly undermined by the effects of noise and outliers, hence the amount of outlier and noise removal required to make the method work. Data from real sources, such as the data in the Basel Dataset, can be of an arbitrary shape (without specific parameters) and often contains noise.

There are two parameters required for the DBSCAN algorithm to work. The first is **epsilon** (*eps*) which is representative of the specified radius between neighbours of core points. If the distance is less than or equal to *eps* then the core points can be considered neighbours, and **could** belong to the same cluster but this is determined by their local density, the number of neighbours within *eps*. This local density is represented in the second parameter which is **minimum samples** (*minSamples*) which states the minimum number of points which are required to form a region. For this project, the library *sklearn.cluster* was used to import the DBSCAN algorithm, and initially the *eps* was set to **3.8** and the *minSamples* was set to **5**. These values were picked by visually by repeatedly running the DBSCAN algorithm to get the least amount of noise points. Below is this *'optimised'* DBSCAN Clustering of the Basel Dataset.

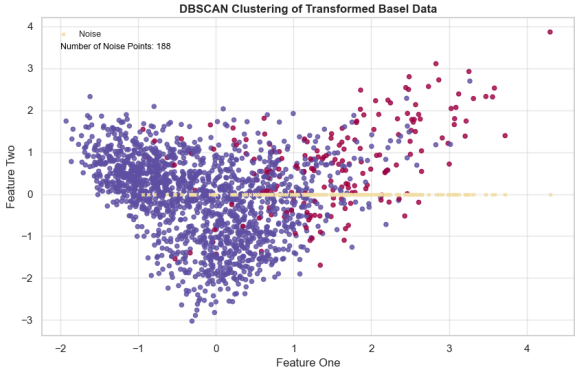

Fig. 7. DBSCAN Clustering of Transformed Basel Data.

Similarly to the *'Elbow'* Method for K-Means Clustering, there is a method to calculate the optimal *eps* value.

A data point x_i is considered:

- **Core point**: A point x_i is a core point if the number of points within its ε-neighborhood (including itself) is greater than or equal to minSamples:

$$\text{Core}(x_i) = |\{x_j \mid \|x_i - x_j\| \leq \varepsilon\}| \geq \text{minSamples}$$

The optimal ε is chosen as:

$$\varepsilon_{\text{optimal}} = d_k(x_i) \quad \text{at the elbow point of the k-distance graph}$$

The number of noise points N_{noise} can be calculated as:

$$N_{\text{noise}} = \sum_{i=1}^{n} \mathbb{I}_{\{\text{label}(x_i)=-1\}}$$

where $\mathbb{I}$ is the indicator function that returns 1 if the condition is true, and 0 otherwise.

When these were ran on the dataset it found that the optimal *eps* value was **5.964811689426372** and the number of noise points was **12**. The number of noise points is significantly less than the number of noise points present within Figure 7. The graph below is the DBSCAN algorithm which has been re-ran on the transfored Basel Climate Data.

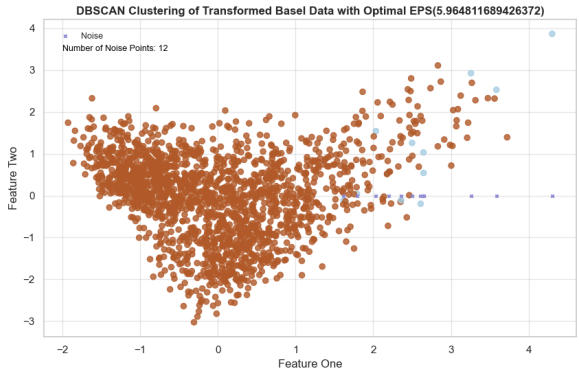

Fig. 8. DBSCAN Clustering of Transformed Basel Data with Optimal EPS.

When taking into account Figures **6**, **7**, and **8** it is clear that these methods of clustering are affective to visualise the Basel Climate Data. There are still outliers present within these Figures and this is indicative that the methods for outlier removal and standardisation could be improved which could generate more accurate results. This should not undermine the results within this report, they are still indicative of clustering relationships within the dataset.

Arguably the greatest weakness with this method of clustering and processing is that it is hard to draw individual effects from the graphs, the clusters show a relationship but they do not actually say what classes these clusters are which makes it difficult to utilise this in the real-world.

IV. Pre-Processing and Feature Extraction

In this task, the aim to use a pre-trained DNN to process images. There were two datasets used for this task, these being the **OxfordPets-IIIT** and the **iRoads** datasets. The OxfordPets-IIIT dataset was created by [1] and the iRoads dataset was created by [2]. The OxfordPets-IIIT dataset is a 37 category pet dataset with around 200 images for each class, which results in approximately 7,400 images within the dataset. The iRoads dataset is a 7 category vehicle-dataset with 4,656 images frames.

The chosen DNN for this project is the ***ResNet-18*** this is because it is one of the cheapest DNNs to implement in terms of computational cost. The ResNet models were proposed by [3] and there are five different ResNet models which correspond to different amounts of layers, they are:

- ***ResNet-18***
- ***ResNet-34***
- ***ResNet-50***
- ***ResNet-101***
- ***ResNet-152***

ResNet Models are built upon the idea of residual connection, in that the DNN model learns the residual (*difference*) between the input and ouput. This differs from other DNNs which aim to directly learn the output. These residual connections then allow for the DNN to skip layers which is beneficial during training as it allows the model to avoid layers.

Preprocessing is different for pretrained DNNs which creates a comparison to the preprocessing performed in II. ResNet models take the standard image size used within ImageNet datasets, which is 224x224px. Therefore, the first step of this process is to resize all of the images within the respective datasets to the correct size. Neural Networks use *tensors* which are multi-dimensional arrays and the second step of DNN preprocessing is to convert the images into tensors for ResNet to read. Image normalisation must still take place, and uses the same values for the Mean (0.485, 0.456, 0.406) and Standard Deviation (0.229, 0.224, 0.225). It follows the same logic as referenced in II and therefore does not need reiterating. Due to use of the *PyTorch* library, the DNNs often require batches of images, and as such the final step of the pre-processing process is to add a singular batch dimension.

V. Clustering

For the clustering of the ResNet-18 DNN, I aimed to incorporate as much as I had already implemented during III. I implemented K-Means Clustering on the extracted features from the model, here the clustering model is assigning features to one of the potential clusters. The measure of Accuracy was used when the ground-truth value was available, and this allowed for the evaluation of K-Means performance. I also re-utilised the PCA algorithm used in III as this allowed for the reduction of dimensionality down to a point where in which it could be visualised on a 2D scatter-plot. Further evaluation of the clustering was provided by the use of the Davies-Bouldin Index which measures average similarity of each cluster tha is most similar to it, and the lower the Davies-Bouldin Index is, the better the clustering must be. The Davies-Bouldin Index is calcuated using the formula:

$$DBI = \frac{1}{N}\sum_{i=1}^{N}\max_{i\neq j}\left(\frac{S_i + S_j}{d(c_i, c_j)}\right) \tag{8}$$

Where:

- N = Number of Clusters
- S_i = Scatter of Cluster i
- $d(c_i, c_j)$ = Euclidean Distance between Centroids and Clusters

Below is the PCA plot from the iRoads dataset which used the ResNet-18 DNN.

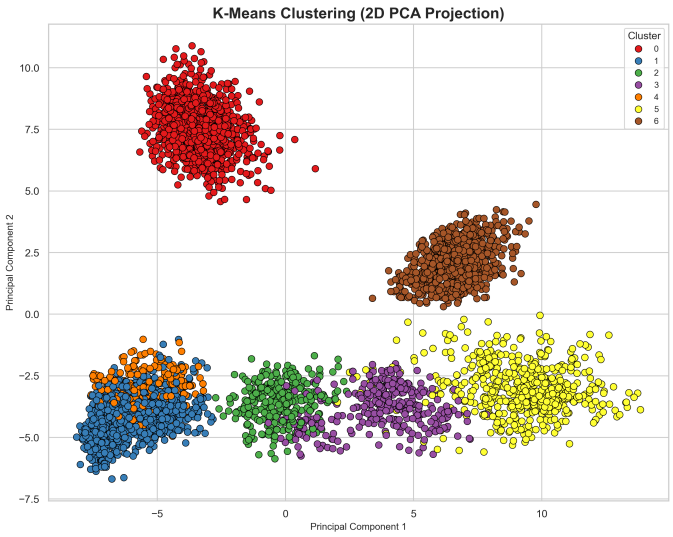

Fig. 9. K-Means Clustering (2D PCA Projection).

VI. Classification

The final output of a DNN is known as the Fully Connected Layer (FCL). The FCL represents the flattening of all of the previous blocks of the network down to one dense, fully connected one-dimensional vector. There are four key areas of Classification within the DNN process. The first refers to the application of the Global Average Pooling (GAP). Within Neural Networks, GAP reduces spatial dimensionality of the network to one-dimension and then feeds that dimension to the FCL. The second area is the FCL stage in which the GAPs one-dimension vector is passed and distributed to the number of classes that were present during the classification task. The classification process uses a function called the ***Softmax*** function. The output of the FCL is a vector of a size $\mathcal{C}$ and the final part of the classification process takes this vector $\mathcal{C}$ and converts using the Softmax function to turn it into a probability distribution wherein the highest class is representative of the index with the highest probability.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{9}$$

Where:

- z_i = Raw Output for Class i
- K = Total Number of Classes
- $d(c_i, c_j)$ = Euclidean Distance between Centroids and Clusters

VII. Conclusions

In conclusion, this report has recounted the Processing of Images and Data with Deep Neural Networks through the completion of two separate, but successive tasks. There is a brief conclusion at the end of III which sums the findings of the first task. However, it should be noted that the first task was not computationally expensive and thus allowed for greater improvement and development.

On the other hand, the training of the ResNet-18 DNN was extremely expensive, computationally. As such, there were only two training epochs for the model where the typical amount is ≥ 10. Overall, the training of the epochs for the DNN took $44\,\mathrm{m}\,22\,\mathrm{s}$ which $= 1268\,\mathrm{s}$ per epoch. Therefore, if this model were to use the standard ten epochs it would take approximately $3\,\mathrm{h}\,31\,\mathrm{m}\,21\,\mathrm{s}$ (*not accounting for variability*).

Below is the recorded **Loss** and **Accuracy** for both Training Epochs:

- Epoch [1/2], Loss: 1.4007, Accuracy: 62.87%
- Epoch [2/2], Loss: 0.6433, Accuracy: 93.92%

A loss of 1.4007 seen in Epoch 1 is moderately highly and suggests that the model is not performing well in this epoch but this is to be expected in the first stage of the DNN Training. This number drops

$$\text{Percentage Change in Loss} = \frac{0.6433 - 1.4007}{1.4007} \times 100 = \frac{-0.7574}{1.4007} \times 100 \approx -54.1\% \tag{10}$$

Change in Loss between Epochs 1 and 2

$$\text{Percentage Change in Accuracy} = \frac{93.92 - 62.87}{62.87} \times 100 = \frac{31.05}{62.87} \times 100 \approx 49.4\% \tag{11}$$

Change in Accuracy between Epochs 1 and 2

As shown above, in 10 and 11, between the first and second epochs that the performance of the model made a significant positive improvement. This drastic improvement between the first and second epochs might also signify that model has insufficient capacity and therefore the number of models layers should increase. Notably, other DNNs such as *VGG-16* only has 16 layers, 13 of which are convolutional and 3 are FCLs, which is suggestive that a fundamental level ResNet-18 is a better model to use.

The overall **Validation Loss**,**Accuracy**, **Precision**, **Recall**, **F1 Score** of the trained model was as follows:

- Validation Loss: 0.4057
- Accuracy: 99.35%
- Precision 0.99
- Recall 0.99
- F1 Score 0.99

The value for the *Validation Loss* was overall quite low which is positive as a higher loss level correlates with worse performance. The value for the *Accuracy* supports the theory that the model is performing well. It is widely accepted that a value $> 90\%$ is indicative that the model is correctly predicting, so a value of 99.35% is very good. The scores for Precision, Recall, and F1 Score indicates that the overall performance of the model is exceptional whilst also being well-balanced due in part to the high precision and recall figures. A future, **definitive** measure would be to test this model on a brand-new, unseen set of test data.

The DBI of the model was **1.7129** which is not a bad score, it is still indicative that the clusters within the model are somewhat overlapped. Further refinement of the Clustering process, possibly implementing another dimensionality reduction method (*i.e.* t-SNE or UMAP) might lower the DBI.

Acknowledgment

The following libraries were used in the development of this report:

- `pandas`
- `numpy`
- `scipy`
- `seaborn`
- `os`
- `matplotlib`
- `sklearn`
  - `decomposition.PCA`
  - `preprocessing.StandardScaler`
  - `ensemble.IsolationForest`
  - `model_selection.train_test_split`
  - `metrics.precision_score`
  - `cluster.KMeans`
  - `cluster.DBSCAN`
  - `neighbors.NearestNeighbors`
  - `metrics.davies_bouldin_score`
  - `metrics.precision_recall_fscore_support`
  - `metrics.confusion_matrix`
- `scipy.cluster`
- `scipy.spatial.distance`
- `PIL` (Pillow)
- `xml.etree.ElementTree`
- `pickle`
- `torch`
- `torch.nn`
- `torch.optim`
- `torch.utils.data.DataLoader`
- `torchvision`
- `tqdm`
- `glob`
- `shutil`
- `random`
- `tarfile`

References

[1] O. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar, "Visual geometry group - university of oxford," Ox.ac.uk, 2024. [Online]. Available: https://www.robots.ox.ac.uk/%7Evgg/data/pets/

[2] M. Rezaei and M. Terauchi, "iroads dataset (intercity roads and adverse driving scenarios)," 2015. [Online]. Available: https://www.cs.auckland.ac.nz/%7Em.rezaei/Publications/iROADS%20Dataset.pdf

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 06 2016.

Appendix

A. IQR for Outlier Detection

```
def IQRforOutliers(df):
    numericalCols = df.select_dtypes(include=['number']).columns
    Q1 = df[numericalCols].quantile(0.25)
    Q3 = df[numericalCols].quantile(0.75)
    IQR = Q3 - Q1
    lowerBound = Q1 - 1.5 * IQR
```

```
    upperBound = Q3 + 1.5 * IQR
    dfNoOutliers = df[~((df[numericalCols] < lowerBound) | (df[numericalCols] > upperBound)).any(axis=1)]
    dfOutliers = df[((df[numericalCols] < lowerBound) | (df[numericalCols] > upperBound)).any(axis=1)]
    return dfNoOutliers, dfOutliers
```

Listing 1. IQR for Outlier Detection

B. Function: PlotScatterAndOutliers

```
def PlotScatterAndOutliers(dfNoOutliers: pd.DataFrame, dfOutliers: pd.DataFrame, baseFilename):

    sns.set_theme(style='whitegrid', palette='pastel')

    numericalCols = dfNoOutliers.select_dtypes(include=['number']).columns

    if len(numericalCols) < 2:
        print("More Numerical Columns required for plot.")
        return

    xCol = numericalCols[0]
    yCol = numericalCols[1]

    plt.figure(figsize=(10,6))
    sns.scatterplot(data=dfNoOutliers, x=xCol, y=yCol, color='deepskyblue', label='Non-Outliers', alpha=.7, s=40)
    sns.scatterplot(data=dfOutliers, x=xCol, y=yCol, color='sandybrown', label='Outliers', marker='X',s=40)

    plt.xlabel(xCol, fontsize=11, labelpad=5)
    plt.ylabel(yCol, fontsize=11, labelpad=5)

    plt.title(f'{xCol} v.s. {yCol}', fontsize=14, weight='bold', pad=10)

    plt.legend(fontsize=9, loc='upper left', frameon=False)
    plt.grid(True, linestyle='--', alpha=.5)
    plt.tight_layout

    filename = get_unique_filename(baseFilename)
    plt.savefig(filename, dpi=300)
    print(f"Plot saved as {filename}")

    plt.show()
```

Listing 2. Plot Scatter Plot with Outliers

C. Function: `normalise`

```
def normalise(data):
    normalisedData = data.copy()

    rows = data.shape[0]
    cols = data.shape[1]

    for j in range(cols):
        maxEl = np.amax(data[:, j])
        minEl = np.amin(data[:, j])

        for i in range(rows):
            normalisedData[i, j] = (data[i, j] - minEl) / (maxEl - minEl)

    return normalisedData
```

Listing 3. Normalizes data to a range of 0 to 1

D. Function: `normalise2`

```
def normalise2(data):
    normalisedData = data.copy()

    rows = data.shape[0]
    cols = data.shape[1]

    for j in range(cols):
        maxEl = np.amax(data[:, j])
        minEl = np.amin(data[:, j])

        for i in range(rows):
            normalisedData[i, j] = -1 + (data[i, j] - minEl) * (1 - (-1)) / (maxEl - minEl)
    return normalisedData
```

Listing 4. Normalizes data to a range of -1 to 1