<u>Part 1:</u>

a)

In the exploratory data analysis, the aim is to explore the MPG dataset and the key or potential relationships between features and the target variable. This included the summary statistics of each explanatory feature, histograms that shows each of their distributions, scatter plots showing relationships between variables and the target and a pairwise plot showing the relationship between each explanatory variable.

In the histograms, it appears that the acceleration feature is the only one that is normally distributed while the rest of the numerical features are skewed to the right. This means that acceleration could be used as a non-transformed feature that doesn't directly impact the bias of the regression unlike the others where transformations may be required. The other two histograms are ordinal categorical features; model_year and cylinders, which shows the relative frequency of each distinct category. It appears that there is a major spike for 4 cylinders in the cylinders histogram, indicating highest frequency (mode) while the model_year frequency is more uniform.

In the scatterplots, negative correlations were observed between mpg and variables like horsepower, weight, and displacement. Heavier and more powerful cars generally have lower fuel efficiency. However, there was a positive relationship between acceleration and mpg, which makes sense since newer cars tend to have faster-accelerating engines which in turn will have a higher mpg value.

In the pairwise plot, we get to see the relationships between each explanatory variable and how they are correlated to each other. We can see that some variables have strong relationships. For example, heavier cars tend to have larger engines and more power, which makes sense. There's also a clear pattern where cars fall into distinct groups based on engine configuration, like 4, 6, or 8 cylinders. On the other hand, some features don't seem to have much of a connection with the rest, like acceleration, which is scattered pretty evenly. Model year also looks fairly spread out, but we do notice that newer cars are generally a bit lighter and possibly more efficient. Overall, these visual patterns help us understand what might influence fuel efficiency and which variables could be more useful in predicting it.

b)

To prepare the dataset for modeling, we first inspected it for missing values. We found that the horsepower feature had six missing entries, accounting for roughly 1.51% of the data. Although a small proportion, handling these missing values is important to maintain consistency and avoid errors during model training. Rather than removing the rows, we chose to impute the missing values using the median of all the numerical columns, which includes horsepower with the missing values. Median imputation is a robust method that helps minimize the influence of outliers, making it a good choice for numerical data like horsepower. For good practice, we also applied imputation for categorical variables using mode in case of any missing values from a category.

In addition to addressing missing data, we also performed several other preprocessing steps. The name column was dropped because it contained text-based identifiers unique to each row and wouldn't contribute meaningful information to the model. For categorical variables, particularly origin, we applied one-hot encoding to convert them into a numerical format, which is required by most machine learning models. We used drop_first=True to avoid multicollinearity by excluding one category as a reference. Finally, we used built-in tools like SimpleImputer to apply these transformations cleanly and efficiently across both the training and test sets, ensuring that the data fed into the linear regression model was properly formatted and ready for analysis.

c)

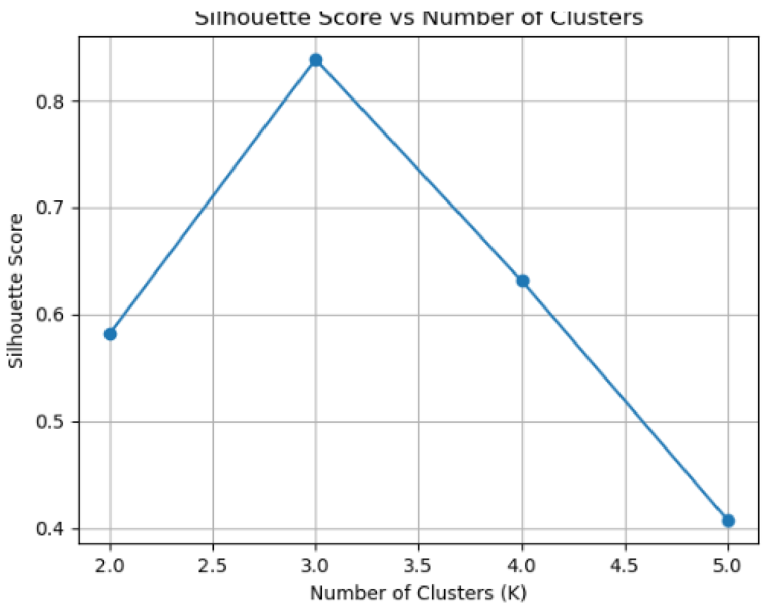
The linear regression model demonstrated strong performance in predicting miles per gallon (MPG), with an R^2 value of 0.8194 on the training set and an even slightly higher R^2 of 0.8339 on the test set. This indicates that over 80% of the variance in MPG can be explained by the selected features, suggesting a well-fitted model that generalizes effectively to unseen data. The mean squared error (MSE) values; 10.78 on the training set and 10.64 on the test set are relatively low, reinforcing the idea that the model is making accurate predictions with minimal average error. Importantly, the similar performance on both training and test sets also suggests that the model is not overfitting.

Looking at the learned coefficients, model year appears to have the strongest positive impact on MPG, reflecting how newer vehicles tend to be more fuel-efficient. Conversely, vehicles from the USA have a strongly negative coefficient, implying they are generally less fuel-efficient compared to the baseline (which would be Europe if one-hot encoding dropped the first level). Features like weight and horsepower are also negatively associated with MPG, which aligns with expectations, as heavier and more powerful cars typically consume more fuel. These insights help explain how different attributes influence fuel efficiency and validate the model's alignment with real-world automotive trends.

Part 2:

a)

To evaluate the effectiveness of clustering in our unsupervised learning model, we applied the K-Means algorithm and used the silhouette score as the primary metric for assessing clustering quality. The silhouette score measures how similar an object is to its own cluster compared to other clusters, with values ranging from -1 (poor clustering) to 1 (well-defined clusters). Our K-values ranged from 2-5. This silhouette score is based on our randomly generated dataset, which was constructed by using make_blobs from scikit_learn which had four features and 3 clusters.

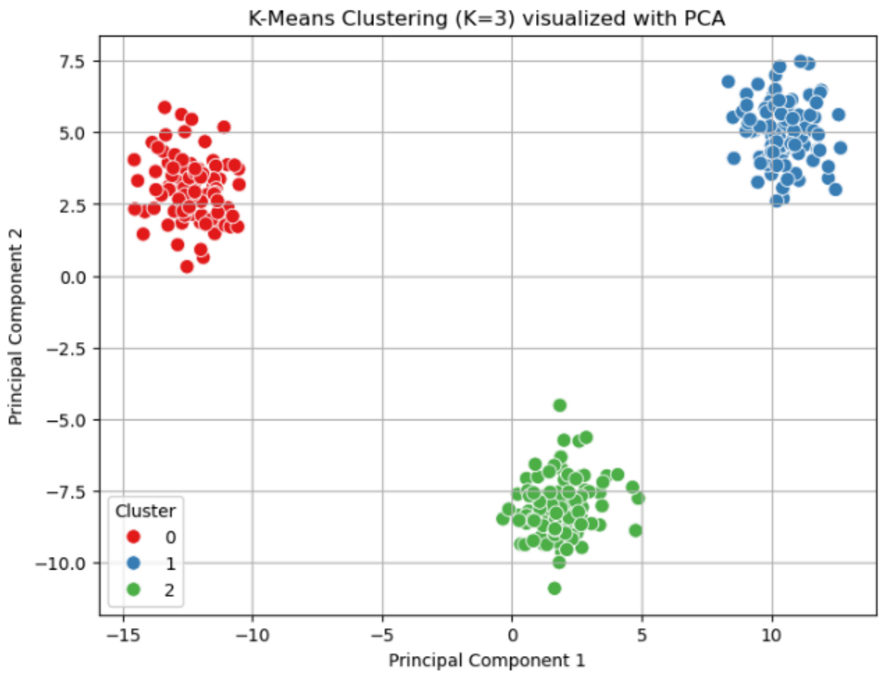


```
K = 2, Silhouette Score = 0.5815
K = 3, Silhouette Score = 0.8383
K = 4, Silhouette Score = 0.6313
K = 5, Silhouette Score = 0.4074
```

Based on the silhouette score and the accompanying line graph, the best K-value is 3.

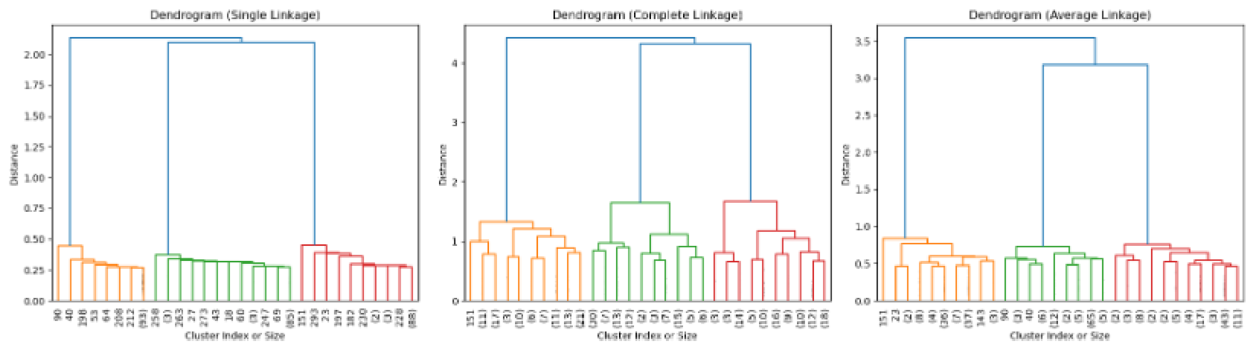
b)

To visualise the clusters identified by the K-Means algorithm, we reduced the dimensionality of the dataset using Principal Component Analysis (PCA). PCA is a linear dimensionality reduction technique that transforms the data into a new coordinate system such that the greatest variance lies on the first principal component, the second greatest on the second component, and so on. Here is the visual representation of the clusters post dimension reduction.



c)

The dendrograms generated using different linkage methods; single, complete, and average, reveal distinct clustering behaviors and structures within the dataset (To simplify the visuals, I used a sample of 30 data points for each linkage method).



The single linkage method, which merges clusters based on the minimum pairwise distance between points, tends to produce elongated and loosely connected clusters due to the “chaining effect”. This can be seen in the dendrogram by the relatively small distances at which many clusters are merged, resulting in less distinct cluster boundaries.

In contrast, complete linkage, which uses the maximum pairwise distance, favours compact and well-separated clusters. Its dendrogram displays taller vertical lines near the top, indicating that clusters are only merged when they are substantially far apart, leading to clearer separation.

The average linkage method offers a balance between the two extremes. It computes the average distance between all pairs of points in two clusters and results in more balanced clusters. The corresponding dendrogram typically shows a moderate merging structure with reasonably distinct cluster separations.

Overall, complete linkage appeared to produce the most well-separated and interpretable clusters for this dataset, while single linkage was more susceptible to chaining. These differences highlight the importance of selecting an appropriate linkage strategy based on the desired clustering characteristics and the nature of the data.

d)

In this scenario, hierarchical clustering and K-Means each offer unique benefits. Hierarchical clustering does not require specifying the number of clusters upfront and provides a clear visual structure through dendrograms, which is useful for understanding relationships between clusters. It also works well for detecting non-spherical clusters. However, it is computationally expensive and can be sensitive to noise and the choice of linkage method.

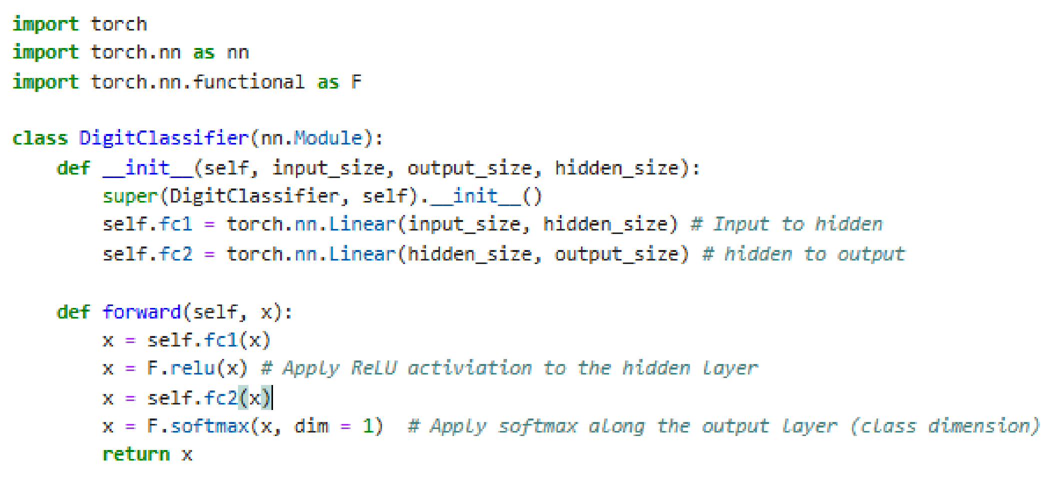
K-Means, on the other hand, is fast and efficient, making it suitable for large datasets. It performed well on this dataset, where clusters were roughly spherical and well-separated. Its main limitations are the need to predefine the number of clusters and its sensitivity to initial centroid placement and outliers.

Overall, hierarchical clustering is better suited for exploratory analysis, while K-Means is more practical for quick, well-structured clustering when the number of clusters is known.

Part 3:

a)

We define a neural network by extending torch.nn.Module with an architecture consisting of one input layer, one hidden layer with 128 neurons using the ReLU activation function, and one output layer with 10 neurons using softmax. The input layer has 64 neurons to match the 8×8 pixel input images flattened into vectors of length 64, while the output layer has 10 neurons to represent the 10 digit classes (0–9). ReLU is chosen for its efficiency in training deep networks by mitigating vanishing gradients, and softmax is used to convert output scores into class probabilities. This simple yet effective structure is suitable for the digit classification task. Here's the implementation of the neural network accordingly:



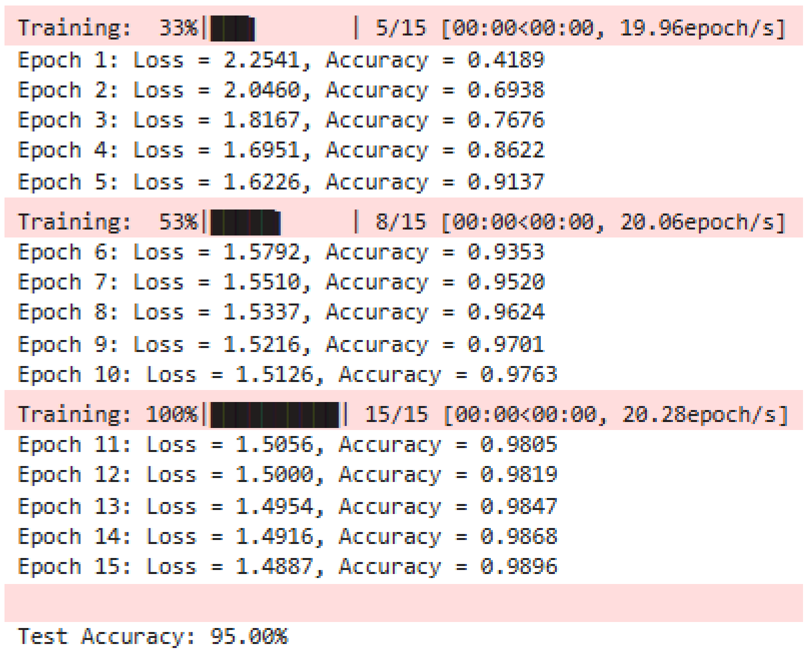
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class DigitClassifier(nn.Module):
    def __init__(self, input_size, output_size, hidden_size):
        super(DigitClassifier, self).__init__()
        self.fc1 = torch.nn.Linear(input_size, hidden_size) # Input to hidden
        self.fc2 = torch.nn.Linear(hidden_size, output_size) # hidden to output

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x) # Apply ReLU activiation to the hidden layer
        x = self.fc2(x)
        x = F.softmax(x, dim = 1)  # Apply softmax along the output layer (class dimension)
        return x
```

b)

For training the neural network, we selected torch.nn.CrossEntropyLoss as the loss function, which is well-suited for multi-class classification tasks like digit recognition. Among the optimiser options, we chose torch.optim.Adam due to its adaptive learning rate capabilities, which adjust the learning rate for each parameter individually based on estimates of first and second moments of the gradients. This typically results in faster convergence and better performance compared to standard SGD, especially on moderately complex models like ours. We set the learning rate to 0.001, a commonly effective default value for Adam that balances learning speed and stability. The model was trained for 15 epochs, and after each epoch, the training loss and accuracy were printed to monitor performance and ensure the model was learning effectively over time.

c)

After training the model and then evaluating its accuracy on the test set, here are the results of its performance:

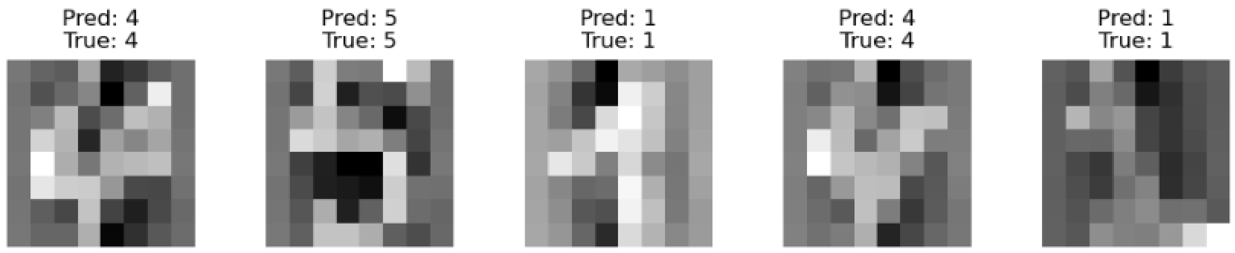


```
Training:  33%|███       | 5/15 [00:00<00:00, 19.96epoch/s]
Epoch 1: Loss = 2.2541, Accuracy = 0.4189
Epoch 2: Loss = 2.0460, Accuracy = 0.6938
Epoch 3: Loss = 1.8167, Accuracy = 0.7676
Epoch 4: Loss = 1.6951, Accuracy = 0.8622
Epoch 5: Loss = 1.6226, Accuracy = 0.9137
Training:  53%|█████     | 8/15 [00:00<00:00, 20.06epoch/s]
Epoch 6: Loss = 1.5792, Accuracy = 0.9353
Epoch 7: Loss = 1.5510, Accuracy = 0.9520
Epoch 8: Loss = 1.5337, Accuracy = 0.9624
Epoch 9: Loss = 1.5216, Accuracy = 0.9701
Epoch 10: Loss = 1.5126, Accuracy = 0.9763
Training: 100%|██████████| 15/15 [00:00<00:00, 20.28epoch/s]
Epoch 11: Loss = 1.5056, Accuracy = 0.9805
Epoch 12: Loss = 1.5000, Accuracy = 0.9819
Epoch 13: Loss = 1.4954, Accuracy = 0.9847
Epoch 14: Loss = 1.4916, Accuracy = 0.9868
Epoch 15: Loss = 1.4887, Accuracy = 0.9896

Test Accuracy: 95.00%
```

And after a few test runs, it has also managed to get a test accuracy of up to 98% and as low as 94%.

For the 5 example predictions, I sampled in 5 images from the test set at random and reconstructed the 64-dim vectors back into the 8 by 8 image and then plotted both the model's prediction and their actual labels:

Here are the 5 examples of predictions with their actual labels:

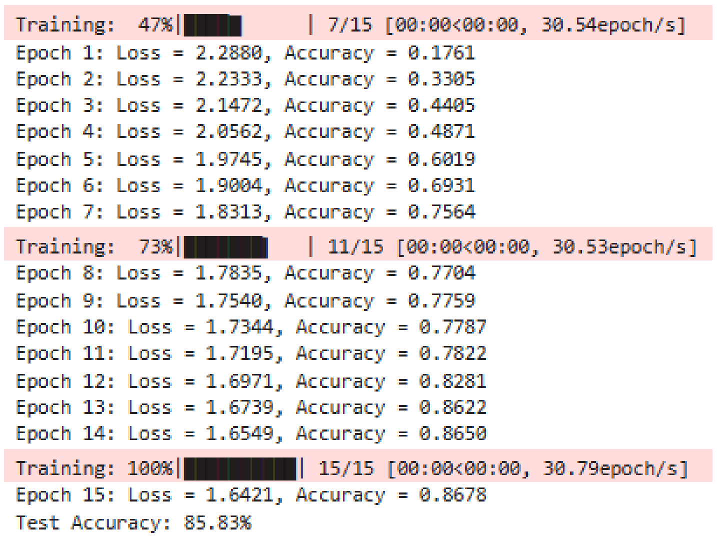


As seen from the example predictions, all five of them have been labelled correctly by the model, which makes sense as the model was able to achieve a 95% accuracy (and even up to 98%).

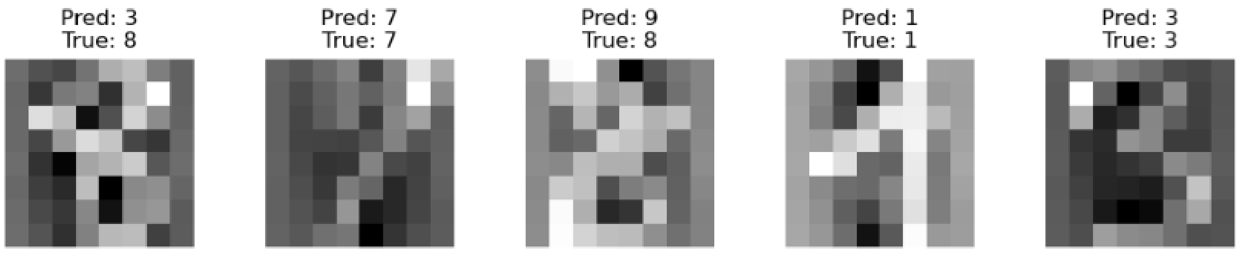
d)

For this part, I simply change the activation function between the input and hidden layers in the NN class and rerun the training of the model. Here are the results of each activation function:

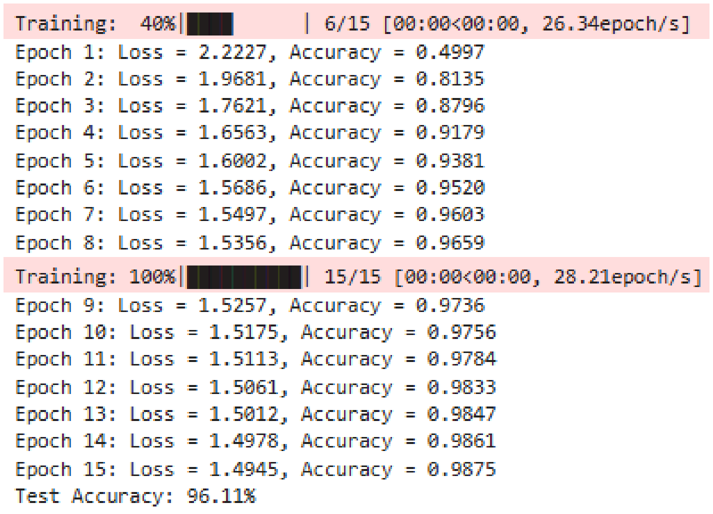
Sigmoid:



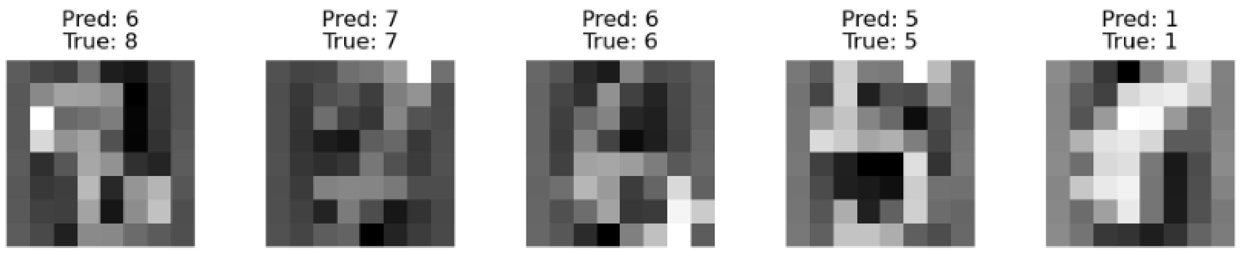
```
Training:  47%|█████     | 7/15 [00:00<00:00, 30.54epoch/s]
Epoch 1: Loss = 2.2880, Accuracy = 0.1761
Epoch 2: Loss = 2.2333, Accuracy = 0.3305
Epoch 3: Loss = 2.1472, Accuracy = 0.4405
Epoch 4: Loss = 2.0562, Accuracy = 0.4871
Epoch 5: Loss = 1.9745, Accuracy = 0.6019
Epoch 6: Loss = 1.9004, Accuracy = 0.6931
Epoch 7: Loss = 1.8313, Accuracy = 0.7564
Training:  73%|███████   | 11/15 [00:00<00:00, 30.53epoch/s]
Epoch 8: Loss = 1.7835, Accuracy = 0.7704
Epoch 9: Loss = 1.7540, Accuracy = 0.7759
Epoch 10: Loss = 1.7344, Accuracy = 0.7787
Epoch 11: Loss = 1.7195, Accuracy = 0.7822
Epoch 12: Loss = 1.6971, Accuracy = 0.8281
Epoch 13: Loss = 1.6739, Accuracy = 0.8622
Epoch 14: Loss = 1.6549, Accuracy = 0.8650
Training: 100%|██████████| 15/15 [00:00<00:00, 30.79epoch/s]
Epoch 15: Loss = 1.6421, Accuracy = 0.8678
Test Accuracy: 85.83%
```



Tanh:



```
Training:  40%|████      | 6/15 [00:00<00:00, 26.34epoch/s]
Epoch 1: Loss = 2.2227, Accuracy = 0.4997
Epoch 2: Loss = 1.9681, Accuracy = 0.8135
Epoch 3: Loss = 1.7621, Accuracy = 0.8796
Epoch 4: Loss = 1.6563, Accuracy = 0.9179
Epoch 5: Loss = 1.6002, Accuracy = 0.9381
Epoch 6: Loss = 1.5686, Accuracy = 0.9520
Epoch 7: Loss = 1.5497, Accuracy = 0.9603
Epoch 8: Loss = 1.5356, Accuracy = 0.9659
Training: 100%|██████████| 15/15 [00:00<00:00, 28.21epoch/s]
Epoch 9: Loss = 1.5257, Accuracy = 0.9736
Epoch 10: Loss = 1.5175, Accuracy = 0.9756
Epoch 11: Loss = 1.5113, Accuracy = 0.9784
Epoch 12: Loss = 1.5061, Accuracy = 0.9833
Epoch 13: Loss = 1.5012, Accuracy = 0.9847
Epoch 14: Loss = 1.4978, Accuracy = 0.9861
Epoch 15: Loss = 1.4945, Accuracy = 0.9875
Test Accuracy: 96.11%
```



Based on the results above, the Sigmoid activation function yielded an average test accuracy of approximately 86%. While respectable, it was notably lower than ReLU. This may be attributed to Sigmoid's tendency to cause vanishing gradients, especially in deeper networks, which can slow down learning and reduce performance. In contrast, the Tanh activation function produced an average test accuracy of 96%, which was on par with ReLU. Tanh provides stronger gradients than Sigmoid due to its zero-centered output, which often leads to better convergence. Overall, while ReLU remains a reliable default choice due to its computational efficiency, Tanh can be a strong alternative depending on the problem context and network structure.

e)

Adjusting the network architecture by adding more hidden layers or changing the number of neurons per layer can significantly impact the model's performance. Increasing the number of hidden layers allows the model to learn more complex and hierarchical patterns in the data, which can lead to improved accuracy, especially on more challenging or non-linear tasks. However, deeper networks also increase the risk of overfitting and require more computational resources and training time. On the other hand, increasing the number of neurons within a layer enhances the network's capacity to capture features, but too many neurons can also lead to overfitting and inefficiency. Conversely, too few neurons may cause the model to underfit, failing to learn the underlying data patterns. Therefore, finding the right balance through experimentation and validation is key to achieving optimal performance.