Customer Churn Prediction Task Report

# Introduction

In this task, we developed an Artificial Neural Network (ANN) model to predict customer churn using various customer attributes like tenure, internet service, and contract type. The process involved several critical steps including data preprocessing, model creation, training, evaluation, and visualization of the results.

# Step-by-Step Explanation

## Step 1: Data Upload

We began by loading the dataset into the environment using Google Colab. This dataset contained various customer details and a binary Churn column indicating whether the customer churned (left the service) or not.

|  |
| --- |
| # Load the dataset (replace 'your\_file.csv' with your actual file name)  uploaded = files.upload()  data = pd.read\_csv('Dataset (ATS)-1.csv') |

## Step 2: Data Preprocessing

In this step, we prepared the dataset for the ANN model. Data preprocessing is crucial as it transforms the raw data into a form that the machine learning model can understand and learn from.

1. Converting Categorical Variables: The Churn column was converted into a binary format where Yes = 1 and No = 0.

|  |
| --- |
| data['Churn'] = data['Churn'].apply(lambda x: 1 if x == 'Yes' else 0) |

2. One-Hot Encoding: Many columns in the dataset, such as gender, InternetService, and Contract, are categorical. These categorical features were converted into numerical values using one-hot encoding to be compatible with the ANN model.

|  |
| --- |
| encoded\_data = pd.get\_dummies(data, columns=['gender', 'Dependents', 'PhoneService', 'MultipleLines',  'InternetService', 'Contract'], drop\_first=True) |

3. Normalization of Numerical Features: Numerical features like tenure and MonthlyCharges were normalized to a scale between 0 and 1 using MinMax scaling. This ensures that no single feature disproportionately influences the model.

|  |
| --- |
| scaler = MinMaxScaler()  encoded\_data[['tenure', 'MonthlyCharges']] = scaler.fit\_transform(encoded\_data[['tenure', 'MonthlyCharges']]) |

## Step 3: Splitting the Data

To evaluate the model performance, the dataset was split into training and testing sets. We used 80% of the data for training the model and 20% for testing it.

|  |
| --- |
| X = encoded\_data.drop('Churn', axis=1)  y = encoded\_data['Churn']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) |

Additionally, the training and testing data were saved and downloaded for further analysis.

|  |
| --- |
| # Save the training and testing datasets with Churn included  train\_data = X\_train.copy()  train\_data['Churn'] = y\_train  train\_data.to\_csv('train\_data\_with\_churn.csv', index=False)  test\_data = X\_test.copy()  test\_data['Churn'] = y\_test  test\_data.to\_csv('test\_data\_with\_churn.csv', index=False)  files.download('train\_data\_with\_churn.csv')  files.download('test\_data\_with\_churn.csv') |

## Step 4: Defining and Training the ANN Model

We defined an Artificial Neural Network (ANN) model using the Keras API in TensorFlow. The model architecture included:

- Input Layer: 10 neurons for 10 features in the dataset.

- First hidden layer with 10 neurons and the ReLU activation function.

- Second hidden layer with 8 neurons, also using the ReLU activation function.

- Output Layer: A single neuron with the sigmoid activation function for binary classification (churn or not churn).

|  |
| --- |
| # Define the architecture of the ANN  model = Sequential()  model.add(Dense(units=10, activation='relu', input\_dim=X\_train.shape[1]))  model.add(Dense(units=8, activation='relu'))  model.add(Dense(units=1, activation='sigmoid'))  # Compile the ANN with Adam optimizer and binary crossentropy loss  model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy']) |

We trained the model for 50 epochs with a batch size of 32 using the Adam optimizer, which is effective for faster convergence in training.

|  |
| --- |
| history = model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_test, y\_test)) |

## Step 5: Model Evaluation

A screen shot of a computer program

Description automatically generated

After training, we evaluated the model on the test dataset to assess its performance. The model achieved a test accuracy of 80.27%, meaning it correctly predicted whether a customer would churn or not for about 80% of the test data.

|  |
| --- |
| test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test)  print(f"Test Accuracy: {test\_accuracy \* 100:.2f}%") |

# Interpretation of the Results

## 1. Test Accuracy (80.27%)

The model's test accuracy is 80.27%, which is a decent performance. It indicates that the model is correctly classifying churn and non-churn customers in about 80% of the cases.

## 2. Loss Plot

A graph of a graph

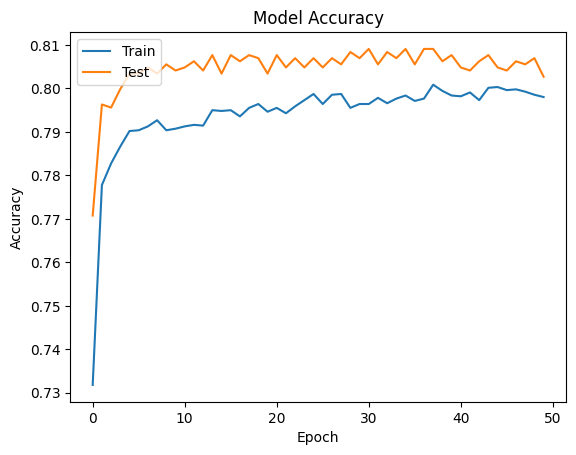
Description automatically generated

The loss plot shows how the training and validation (test) loss decreased over the course of 50 epochs. Here's what we observe:

- Initial High Loss: The loss starts at a higher value but quickly decreases in the first few epochs.

- Stabilization: After around 10 epochs, the training and test loss curves start to flatten. Both curves stabilize at similar levels, indicating that the model is not overfitting and generalizes well to unseen data.

## 3. Accuracy Plot



The accuracy plot shows the progression of accuracy over 50 epochs:

- Initial Accuracy Improvement: The accuracy increases significantly within the first few epochs, indicating that the model quickly learns the patterns in the data.

- Fluctuation in Test Accuracy: The test accuracy curve fluctuates slightly between epochs but generally stays around 80%. This fluctuation is common in models trained on real-world data.

# Conclusion

In this task, we built an ANN model to predict customer churn with an accuracy of **80.27%** on the test data. This accuracy is indicative of a strong model that can identify patterns in customer behavior, helping to predict churn. The stability of the loss and accuracy plots suggests that the model is generalizing well to unseen data, with little overfitting.