

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:

Step 1: Input Layer

The first layer of the ANN is the **input layer**. This layer takes the input features of the dataset. Since we have 10 input features after data preprocessing, the input layer will have 10 neurons.

Explanation:

- **Neurons in the input layer** match the number of features. Each neuron represents one feature from the dataset (e.g., tenure, MonthlyCharges, etc.).

```
model.add(Dense(units=10, activation='relu', input_dim=X_train.shape[1]))
```

This means we are creating an input layer with 10 neurons using the **ReLU activation function** (Rectified Linear Unit), which helps introduce non-linearity into the model.

Step 2: Hidden Layers

- **Hidden layers** help the network learn complex relationships in the data. We used two hidden layers in this model:
 1. The first hidden layer has 10 neurons. First hidden layer with 10 neurons and the ReLU activation function.
 2. The second hidden layer has 8 neurons. Second hidden layer with 8 neurons, also using the ReLU activation function.

Explanation:

- Hidden layers are where the model starts to learn patterns in the data by applying weights to each feature.
- The **ReLU activation function** is commonly used because it helps the model converge faster and solves the problem of vanishing gradients (an issue with deep learning models).

```
• model.add(Dense(units=10, activation='relu'))  
• model.add(Dense(units=8, activation='relu'))
```

Output Layer:

A single neuron with the sigmoid activation function for binary classification (churn or not churn). The **output layer** is the final layer that makes the predictions. Since we are dealing with a binary classification problem (churn or no churn), we have one neuron in the output layer. The **sigmoid activation function** is used here because it outputs probabilities between 0 and 1, which can be interpreted as the likelihood of churn.

```
model.add(Dense(units=1, activation='sigmoid'))
```

The sigmoid function converts the output into a probability, where values close to 1 indicate a higher likelihood of churn and values close to 0 indicate no 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))
```

2. Compiling and Training the Model

Step 4: Compiling the Model

- Once the model is defined, we compile it. Compiling the model means specifying the loss function, optimizer, and metrics that will be used during training.

We use:

- **Binary Crossentropy Loss:** This is used for binary classification tasks and measures the difference between actual and predicted values.
- **Adam Optimizer:** Adam is an efficient gradient-based optimization algorithm that adjusts the learning rate during training.
- **Accuracy:** This metric will help us track how often the model is correctly predicting churn during training.

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Step 5: Training the Model

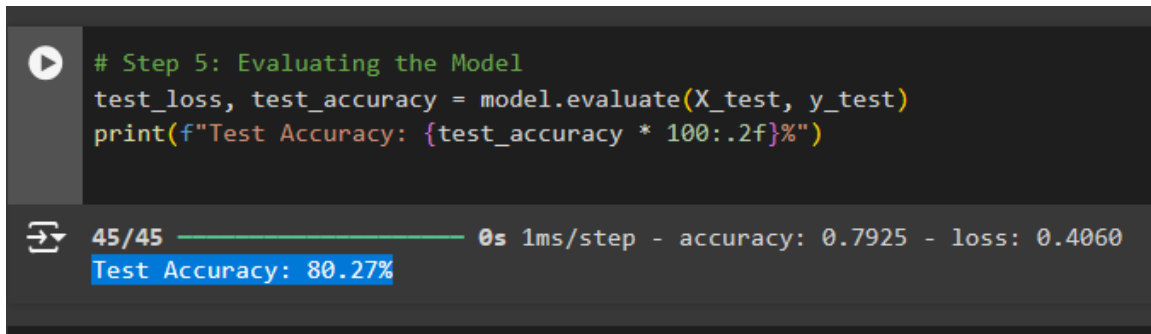
- The model is trained using the **training data** (X_train and y_train). We train the model for 50 **epochs** (one epoch = one complete cycle through the training dataset), with a batch size of 32 (meaning that the data is divided into batches of 32 samples each, and the model updates weights after each batch).

```
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test))
```

Explanation:

- **Training** involves feeding data to the model, calculating the error (loss), and adjusting the model's weights to minimize the error over time.
- **Validation Data:** The test data is used to check how well the model performs on unseen data after each epoch. This helps in detecting overfitting (when a model performs well on training data but poorly on new data).

Step 5: Model Evaluation



```
# Step 5: Evaluating the Model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

45/45 ————— 0s 1ms/step - accuracy: 0.7925 - loss: 0.4060
Test Accuracy: 80.27%

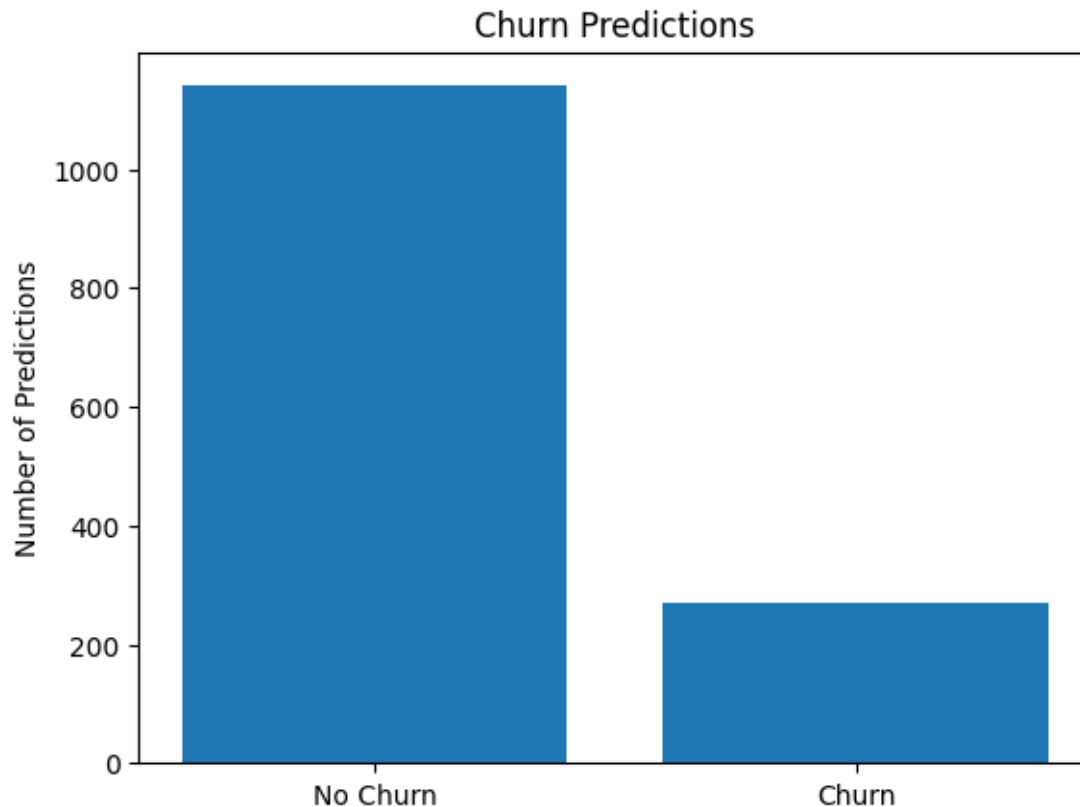
After training, we evaluated the model on the test dataset to assess its performance. The model achieved a test accuracy of 80.27%, meaning it correctly predicted whether a customer would churn or not for about 80% of the test data.

```
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.

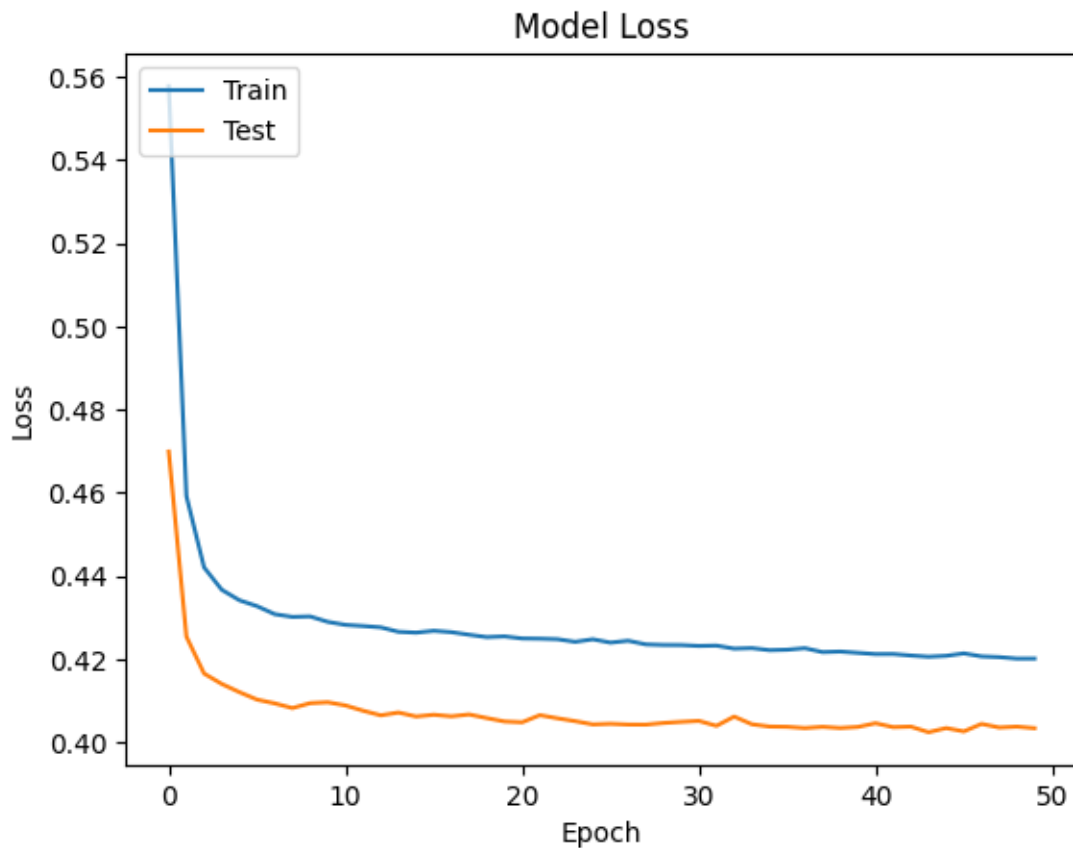


The model is predicting that the majority of customers are unlikely to churn, which is common in many datasets, as often only a small percentage of customers actually churn in real-world scenarios. The difference in the height of the two bars means that most customers in the test set are being classified as "No Churn" by the model, while a smaller portion are predicted as "Churn."

Why This Matters:

1. **Imbalanced Dataset:** If the actual dataset contains many more "No Churn" cases than "Churn" cases, this might be why the model is predicting fewer churn cases. You can check the original distribution of churn vs. no churn in your dataset.
2. **Potential Model Bias:** If the model is biased towards predicting "No Churn" more often, it may be due to an imbalance in the data (more "No Churn" than "Churn" cases) or model training, and additional techniques such as oversampling or undersampling might be needed.

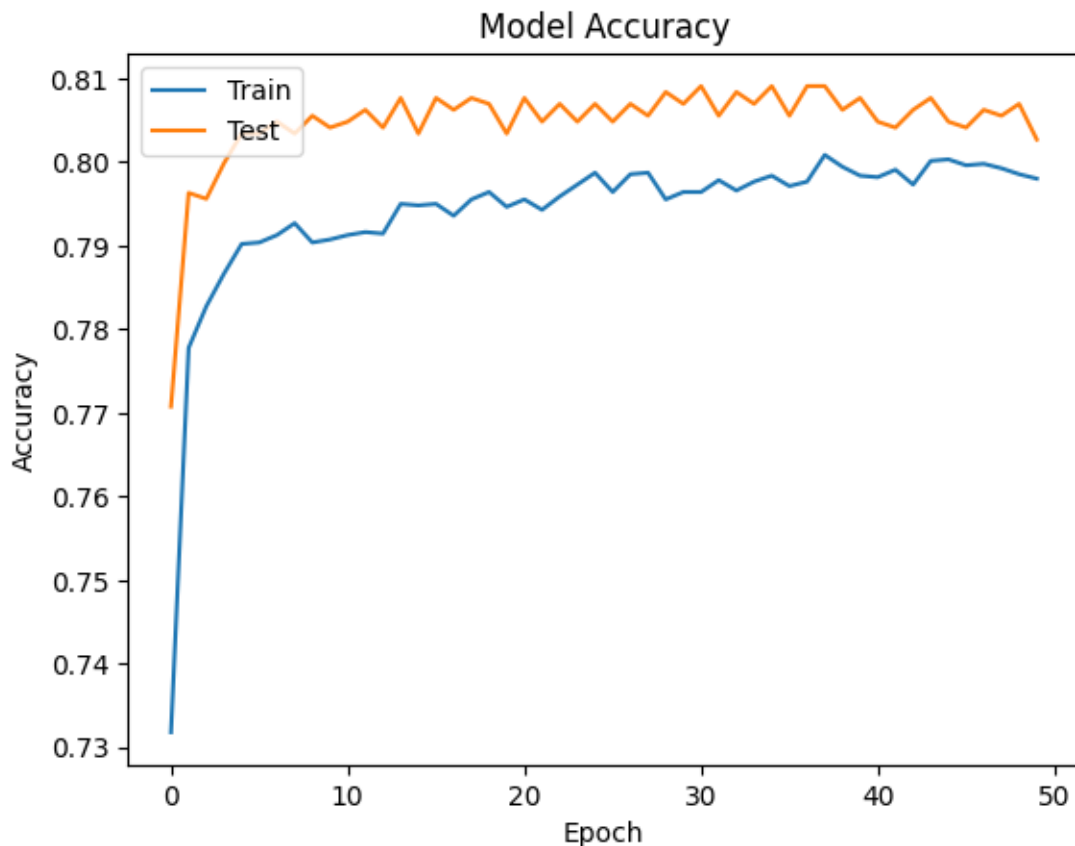
2. Loss Plot



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



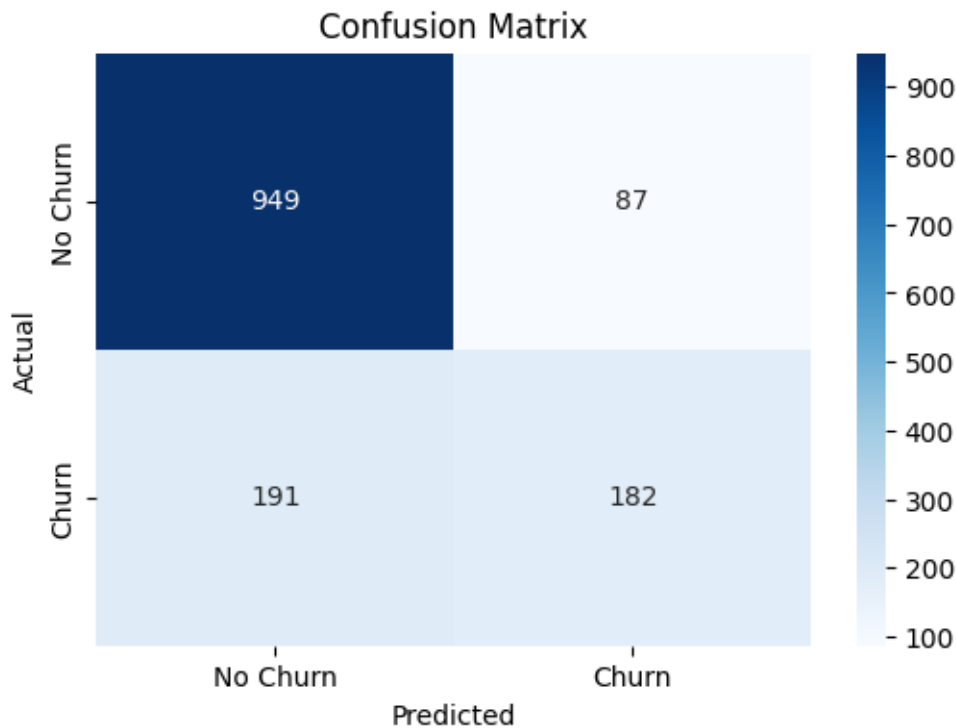
The accuracy plot shows the progression of accuracy over 50 epochs:

- **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.

Next Steps:

- **Evaluate Model Performance Further:** we can check the **confusion matrix** or calculate precision, recall, and F1-score to see if the model is performing well in identifying churned customers accurately.
- **Address Data Imbalance:** If we find that the dataset is heavily imbalanced, we need to consider resampling techniques to address the imbalance between churn and no churn cases.

Confusion matrix



- Precision: 0.68
- Recall: 0.49
- F1-Score: 0.57

1. Confusion Matrix Explanation:

- True Positives (TP): 182 customers were correctly predicted to churn (actual churn and predicted churn).
- True Negatives (TN): 949 customers were correctly predicted not to churn (actual no churn and predicted no churn).
- False Positives (FP): 87 customers were incorrectly predicted to churn (predicted churn but actually no churn).
- False Negatives (FN): 191 customers were incorrectly predicted not to churn (predicted no churn but actually churn).

2. Precision, Recall, and F1-Score:

- Precision: 0.68: Out of all the customers that the model predicted to churn (269 customers), 68% were actually churn cases, meaning the model is relatively accurate in predicting churn, but it does make some incorrect churn predictions (false positives).
- Recall: 0.49: The model identified only 49% of the actual churn cases (373 total actual churns), meaning the model is missing a significant number of churn cases (false negatives).
- F1-Score: 0.57: The F1-score of 0.57 indicates a moderate balance between precision and

recall. The model struggles more with recall (missing churn cases) than with precision (incorrectly predicting churn).

3. Analysis of the Results:

Strengths:

- Good Precision (0.68): The model performs relatively well when it predicts churn. Out of all the customers it predicts to churn, 68% are indeed churners, meaning there are fewer false alarms.
- True Negatives: The model correctly identifies most of the "No Churn" cases (949 out of 1036), which indicates that it performs well in predicting customers who will stay.

Weaknesses:

- Low Recall (0.49): The model misses a significant number of actual churn cases (false negatives), which could lead to missed opportunities for retention strategies.
- False Negatives (191 cases): These are customers who actually churned but were predicted not to churn by the model. This could be concerning if the goal is to retain churned customers.

Key Insights:

- Imbalanced Performance: The model does well in identifying customers who will not churn, but it struggles to correctly identify all customers who are at risk of churning (low recall).
- Implications for Retention Strategies: While the model may provide useful predictions for a subset of churners (68% precision), it may miss many customers who are at risk, leading to incomplete retention efforts.
- Focus on Improving Recall: If your goal is to reduce customer churn, you'll want to focus on improving recall by capturing more of the actual churn cases.

4. Recommendations:

- Improve Recall: Since customer churn is often a critical business metric, improving recall should be a priority. Techniques include adjusting the decision threshold or using resampling methods to ensure the model captures more churn cases.
- Balance Precision and Recall: Depending on the business needs, you may want to balance precision (avoiding false positives) and recall (capturing as many true churns as possible). The F1-score will help measure this balance.
- Consider the Business Impact: Missing actual churn cases (low recall) could result in

missed opportunities for customer retention. Focus on identifying more churners, even at the cost of slightly lower precision.

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.