Architecture of the ANN Model

Defined Architecture of ANN Model

The architecture of an ANN model includes the configuration of its layers, which are crucial in determining how well the model can learn from the data and make accurate predictions. Below is a detailed explanation of each part of the architecture:

Input Layer

The input layer is the first layer of the ANN model, which receives the input data. It is defined by the shape of the input data, which corresponds to the number of features in the dataset. In this case, we use the Input function to specify the input shape.

Code

```
# Define input shape based on the number of features
input_shape = (X_train_scaled.shape[1],)
input_layer = Input(shape=input_shape)
```

Hidden Layers

Hidden layers are where the ANN model performs most of its computations. These layers consist of neurons (units) that apply transformations to the input data. The ReLU (Rectified Linear Unit) activation function is commonly used in hidden layers to introduce non-linearity into the model, allowing it to learn complex patterns.

Code

```
# First hidden layer with 64 neurons
hidden_layer_1 = Dense(units=64, activation='relu')(input_layer)

# Second hidden layer with 32 neurons
hidden_layer_2 = Dense(units=32, activation='relu')(hidden_layer_1)

# Third hidden layer with 16 neurons
hidden_layer_3 = Dense(units=16, activation='relu')(hidden_layer_2)
```

Output Layer

The output layer is the final layer of the ANN model, which provides the model's predictions. For binary classification tasks, such as predicting customer churn, the output layer has a single neuron with a sigmoid activation function, which outputs a probability value between 0 and 1.

Code

```
# Output layer for binary classification
output_layer = Dense(units=1, activation='sigmoid')(hidden_layer_3)
```

Model Compilation

After defining the layers, the model is compiled. Compilation involves specifying the optimizer, loss function, and metrics. The Adam optimizer is an adaptive learning rate optimization algorithm that is widely used for training neural networks. Binary cross-entropy is used as the loss function for binary classification tasks, and accuracy is used as the evaluation metric.

Code

```
# Compile the ANN model
ann_model = tf.keras.Model(inputs=input_layer, outputs=output_layer)
ann_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Trained ANN Model on Provided Dataset

Training the ANN model involves feeding the data into the model, allowing it to learn the patterns, and evaluating its performance. Here's a detailed step-by-step explanation:

Data Preparation

First, the dataset is loaded, and categorical variables are encoded. The data is then split into training and testing sets, and the features are scaled.

Code

```
# Load and preprocess the dataset
df = pd.read_csv('Dataset (ATS).csv')

# Encode categorical variables
categorical_cols = ['gender', 'Dependents', 'PhoneService', 'MultipleLines',
'InternetService', 'Contract', 'Churn']
le = LabelEncoder()
for col in categorical_cols:
    df[col] = le.fit_transform(df[col])

# Split the dataset into features (X) and target (y)
X = df.drop('Churn', axis=1)
y = df['Churn']

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Model Training

The ANN model is trained using the training data. Early stopping is implemented to prevent overfitting by stopping the training process if the validation loss does not improve after a specified number of epochs (patience).

Code

```
# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
```

```
# Train the model with early stopping
history = ann_model.fit(X_train_scaled, y_train, epochs=50, batch_size=32,
validation_split=0.2, verbose=1, callbacks=[early_stopping])
```

Model Evaluation

After training, the model's performance is evaluated using the test data. The model makes predictions on the test data, and metrics such as accuracy and a classification report are used to assess its performance.

Code

```
# Predict churn on the test data
y_pred = (ann_model.predict(X_test_scaled) > 0.5).astype(int)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
classification_report_output = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print(f"Classification Report: \n{classification_report_output}")
```

Saving the Trained Model

The trained ANN model is saved to a file for future use. This allows you to load and use the trained model without having to retrain it.

Code

```
# Save the trained model
ann_model.save('trained_ann_model.h5')
```