Mohammad Idrees Bhat

Tech Skills Trainer | AI/ML Consultant

Building Neural Networks

Constructing and training neural networks

AGENDA

1. Neural Networks
2. Creating a Neural network

If you could have dinner with any historical figure, who would it be and why?

1. Neural Networks

- Intro to Neural Networks by 3B1B What are Neural Networks

What is a Neural Network?

A **neural network** is a computational model inspired by the way biological neural networks in the human brain process information. It consists of layers of nodes (also known as neurons), where each node in a layer is connected to every node in the previous and next layers.

The primary goal of a neural network is to learn a mapping from inputs to outputs by adjusting the weights of the connections between neurons based on the data.

Let's start with a simple neural network!

Understanding the Structure of a Neural Network

A neural network consists of three types of layers:

- **Input Layer**: The layer where the data is fed into the network. It corresponds to the features of the data.
- **Hidden Layers**: Intermediate layers between the input and output layers. These layers perform computations using weights and activation functions.
- **Output Layer**: The final layer that produces the network's predictions. For binary classification, it usually has a sigmoid activation function, which outputs a value between 0 and 1.

Visual Representation of a Neural Network

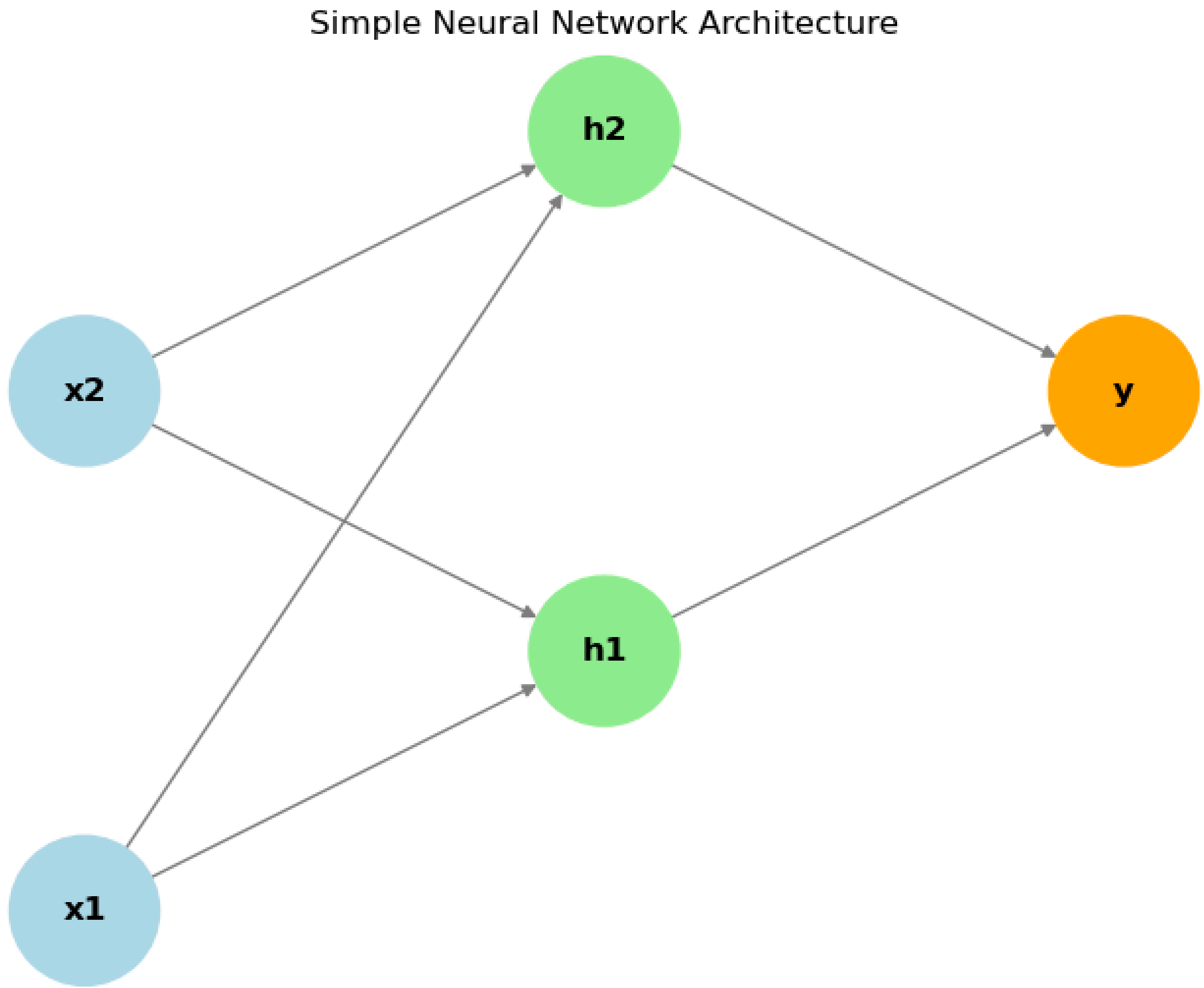
Here’s a simple visualization of a neural network with an input layer, one hidden layer, and an output layer:

Display a simple neural network diagram using matplotlib or plotly.

This code will display a simple neural network diagram with two input nodes, two hidden nodes, and one output node. The graph is clear and color-coded for better understanding.

In [37]:
```
import matplotlib.pyplot as plt
import networkx as nx

def plot_simple_nn():
    # Create a directed graph
    G = nx.DiGraph()

    # Nodes: Input Layer (x1, x2), Hidden Layer (h1, h2), Output Layer (y)
    input_nodes = ['x1', 'x2']
    hidden_nodes = ['h1', 'h2']
    output_node = ['y']

    # Add nodes to the graph with their layers
    G.add_nodes_from(input_nodes, layer='input')
    G.add_nodes_from(hidden_nodes, layer='hidden')
```

```python
    G.add_nodes_from(output_node, layer='output')

    # Add edges (connections between nodes)
    for node_in in input_nodes:
        for node_hidden in hidden_nodes:
            G.add_edge(node_in, node_hidden)

    for node_hidden in hidden_nodes:
        G.add_edge(node_hidden, output_node[0])

    # Set node colors by layer
    node_colors = ['lightblue' if G.nodes[node]['layer'] == 'input' else
                   'lightgreen' if G.nodes[node]['layer'] == 'hidden' else
                   'orange' for node in G.nodes]

    # Define positions of nodes for a clear layout
    pos = {'x1': (0, 0), 'x2': (0, 1), 'h1': (1, 0.5), 'h2': (1, 1.5), 'y': (2, 1)}

    # Draw the graph with the specified positions
    nx.draw(G, pos, with_labels=True, node_size=3000, node_color=node_colors, font_size=12, font_weight='bold', edge_color='gray')

    # Set the title of the plot
    plt.title("Simple Neural Network Architecture")
    plt.show()

# Call the function to plot the neural network
plot_simple_nn()
```




```
In [ ]:
```

How Does a Neural Network Learn?

Neural networks learn by adjusting their **weights**. Initially, these weights are random, and the network's output is inaccurate. The learning process involves training the network using data to minimize the **loss function**.

The learning algorithm that neural networks use is called **Backpropagation**. Here's a high-level overview of the process:

1. **Forward Propagation**: The input data passes through the network, from the input layer to the output layer, making predictions.
2. **Calculate Loss**: The prediction is compared with the actual target (the true label), and the error (loss) is calculated.
3. **Backpropagation**: The loss is propagated backward through the network, adjusting the weights of the connections to reduce the error.

4. **Optimization**: The weights are updated using an optimization algorithm (e.g., **Gradient Descent**).

The network repeats this process multiple times over many epochs until the loss is minimized.

Activation Functions

Activation functions are mathematical functions used in the hidden and output layers of a neural network to introduce non-linearity. They help the model learn complex patterns.

Common Activation Functions:

- **ReLU (Rectified Linear Unit)**: Most commonly used in hidden layers. It outputs the input directly if it's positive; otherwise, it outputs zero.
- **Sigmoid**: Used in the output layer for binary classification. It maps values to a range between 0 and 1.
- **Tanh**: Similar to sigmoid but maps values to the range between -1 and 1.

Why Non-Linearity?

Without non-linearity (activation functions), a neural network would just be a linear model, no matter how many layers it has. Activation functions allow the network to learn complex, non-linear relationships in data.

Building Our First Neural Network with Keras

TensorFlow is an open-source deep learning framework developed by Google.

- https://www.tensorflow.org

It provides a set of tools to build and train machine learning models, particularly those involving neural networks.

Think of TensorFlow as a "workbench" for creating, training, and deploying models.

TensorFlow gets its name from the term "**tensor**", which refers to the multi-dimensional arrays (like matrices or higher-dimensional structures) that are used to represent data in machine learning.

Flow refers to the way data moves through the various operations (or layers) of a model, forming a "**computational graph.**"

In [38]:
```python
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

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

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
Epoch 1/5
c:\Users\devid\anaconda3\Lib\site-packages\keras\src\layers\reshaping\flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` a
rgument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8585 - loss: 0.4854
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9550 - loss: 0.1494
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9692 - loss: 0.1078
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9734 - loss: 0.0847
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9775 - loss: 0.0727
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 826us/step - accuracy: 0.9726 - loss: 0.0871
```

Out[38]:
```
[0.07361076027154922, 0.9764000177383423]
```

What is happening in the model part?

- **Flatten**: Converts the 2D image (28x28) into a 1D list (784 values).
- **Dense (128 neurons)**: Tries to learn the patterns in the image with 128 "neurons."
- **Dropout**: Helps avoid overfitting by randomly ignoring some neurons during training.

- **Dense (10 neurons)**: Decides which digit (0-9) the image represents, with each neuron representing a digit and the highest probability being the chosen one.

```
tf.keras.layers.Flatten(input_shape=(28, 28)),
```

- Flatten is like "unfolding" or "stretching" the 28x28 images into a straight line.
- A 28x28 image has 28 rows and 28 columns of pixels. When we use the flatten layer, we take all those 784 pixels (28 * 28 = 784) and make them into a single row of 784 values.
- This is needed because a Dense layer (next layer) only accepts 1D input (a list of numbers, not a 2D matrix).

```
tf.keras.layers.Dense(128, activation='relu'),
```

- This is a Dense layer, meaning every pixel in the input (784 numbers) is connected to 128 neurons in this layer. Each neuron processes the input in its own way and gives an output.

- Think of this layer as having 128 "neurons" that each try to understand the image in their own way, making it learn complex patterns and features like edges, shapes, etc.

- ReLU (Rectified Linear Unit) is an activation function.

  It just means that if the neuron's output is negative, it becomes zero (this helps the model learn faster and avoid overfitting).

  If it's positive, the output stays the same. It's like saying "if you're not positive enough, you get nothing."

```
tf.keras.layers.Dropout(0.2),
```

- This is a Dropout layer, and it's a trick used during training to help the model learn better.
- During each step of training, it randomly "turns off" 20% of the neurons (0.2 means 20%).
- This prevents the model from becoming too reliant on any one neuron and forces it to learn more general patterns, making it less likely to overfit (become too specific to the training data).
- It's like a student who studies by forcing themselves to learn everything, not just relying on one method.

```
tf.keras.layers.Dense(10, activation='softmax')
```

- This is the output layer, where the final decision happens.
- It has 10 neurons because there are 10 possible digit classes (0 through 9) in the MNIST dataset. Each neuron in this layer corresponds to one digit.
- Softmax is an activation function used here, and it converts the output of these 10 neurons into probabilities (values between 0 and 1) that sum up to 1.

Keras is a high-level neural network API, written in Python.

It is designed to simplify the process of building deep learning models by providing an easy-to-use interface.

Keras allows you to define neural networks with just a few lines of code, abstracting away many of the complexities involved in using lower-level libraries like TensorFlow.

Think of Keras as the user-friendly interface or the "wrapper" around TensorFlow. It makes it easier to experiment and prototype with deep learning models.

Here’s another analogy:

Keras is like a remote control for the TensorFlow toolbox. It makes it simpler to use TensorFlow’s power without dealing with all the fine details.

How do TensorFlow and Keras come together:

- **Building the Model**: You use Keras to define the layers and architecture of your neural network (input, hidden, output layers).
- **Training the Model**: When you call methods like model.fit(), Keras takes care of the details, but under the hood, TensorFlow is handling the data flow and optimization processes.
- **Evaluation and Prediction**: After training, you can use Keras to evaluate the model's performance or make predictions, again leveraging TensorFlow's efficient computation engine.

Now that we have the basic understanding of neural networks, let's build a simple one using **Keras**.

We'll use the **Sequential** model, which is a linear stack of layers.

Steps:

1. **Import necessary libraries**.
2. **Prepare the data** (split into training and testing sets).
3. **Build the model** (define input, hidden, and output layers).
4. **Compile the model** (set optimizer, loss function, and metrics).
5. **Train the model**.
6. **Evaluate the model**.

Let’s start by building and training the model.

In [39]:
```
from keras.models import Sequential
from keras.layers import Dense
```

```python
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

# Generate a simple binary classification dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)

# Split the dataset 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)

# Build the neural network model
model = Sequential([
    Dense(16, activation='relu', input_shape=(X_train.shape[1],)),  # Input layer
    Dense(8, activation='relu'),                                   # Hidden layer
    Dense(1, activation='sigmoid')                                  # Output layer (binary classification)
])

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

# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
Epoch 1/10
```

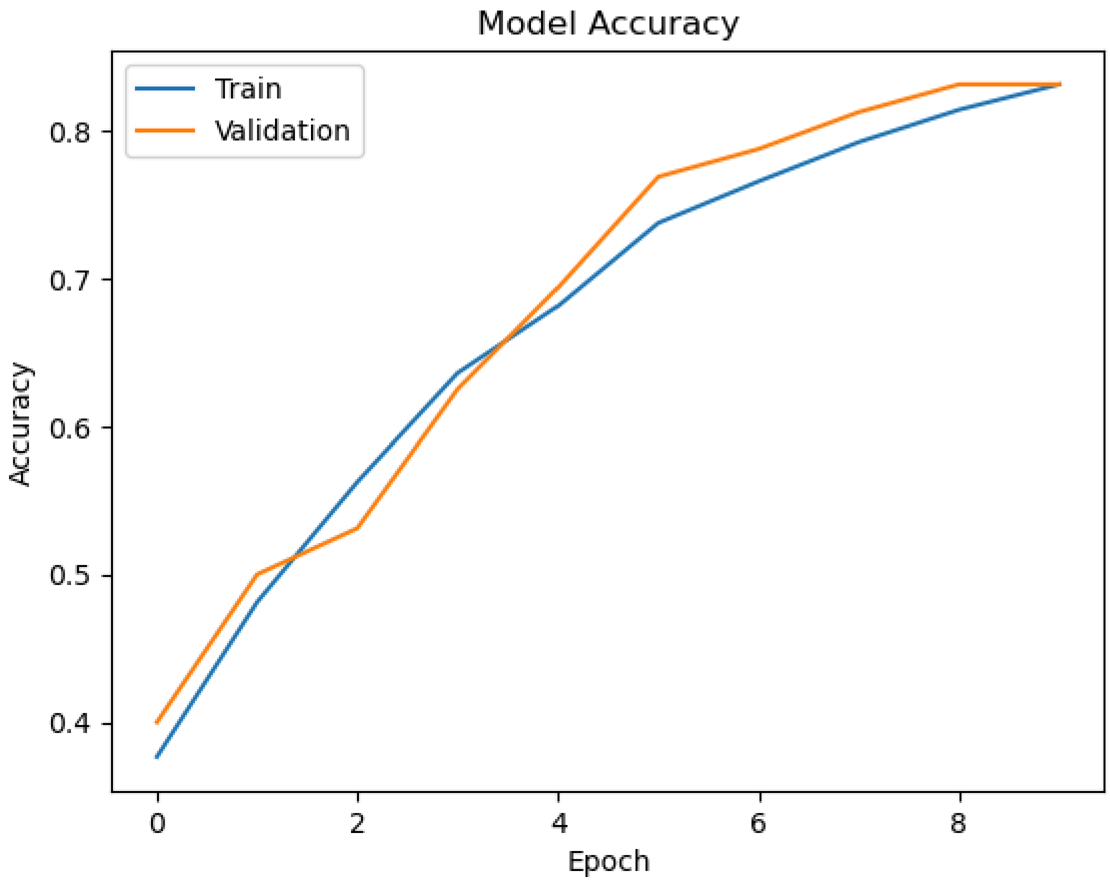
```
c:\Users\devid\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.3547 - loss: 0.8714 - val_accuracy: 0.4000 - val_loss: 0.8245
Epoch 2/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4421 - loss: 0.7731 - val_accuracy: 0.5000 - val_loss: 0.7467
Epoch 3/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5112 - loss: 0.7113 - val_accuracy: 0.5312 - val_loss: 0.6915
Epoch 4/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6440 - loss: 0.6356 - val_accuracy: 0.6250 - val_loss: 0.6479
Epoch 5/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6424 - loss: 0.6205 - val_accuracy: 0.6938 - val_loss: 0.6084
Epoch 6/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7155 - loss: 0.5732 - val_accuracy: 0.7688 - val_loss: 0.5739
Epoch 7/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7572 - loss: 0.5319 - val_accuracy: 0.7875 - val_loss: 0.5417
Epoch 8/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8025 - loss: 0.4997 - val_accuracy: 0.8125 - val_loss: 0.5108
Epoch 9/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8175 - loss: 0.4771 - val_accuracy: 0.8313 - val_loss: 0.4841
Epoch 10/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8524 - loss: 0.4350 - val_accuracy: 0.8313 - val_loss: 0.4561
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7990 - loss: 0.4857
Test Loss: 0.4872265160083771
Test Accuracy: 0.7950000166893005
```

Visualizing Model Performance

To understand how the model is performing over time, let's plot the **training** and **validation** loss and accuracy across epochs.

In [43]:
```python
import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



- Training Accuracy Line: Represents how well the model is doing on the training dataset after each epoch.
- Validation Accuracy Line: Represents how well the model is doing on the validation dataset, which is separate data used to test the model during training.

Evaluating the graph:

- Training and validation lines should follow similar trends.

Both should rise and plateau, with training accuracy slightly higher.

- Validation accuracy should not diverge too much from training accuracy.

A small gap is normal, but a large gap suggests overfitting.

- If validation accuracy plateaus early, consider tuning hyperparameters.

This might involve changing the learning rate, adding layers, or modifying the model architecture.

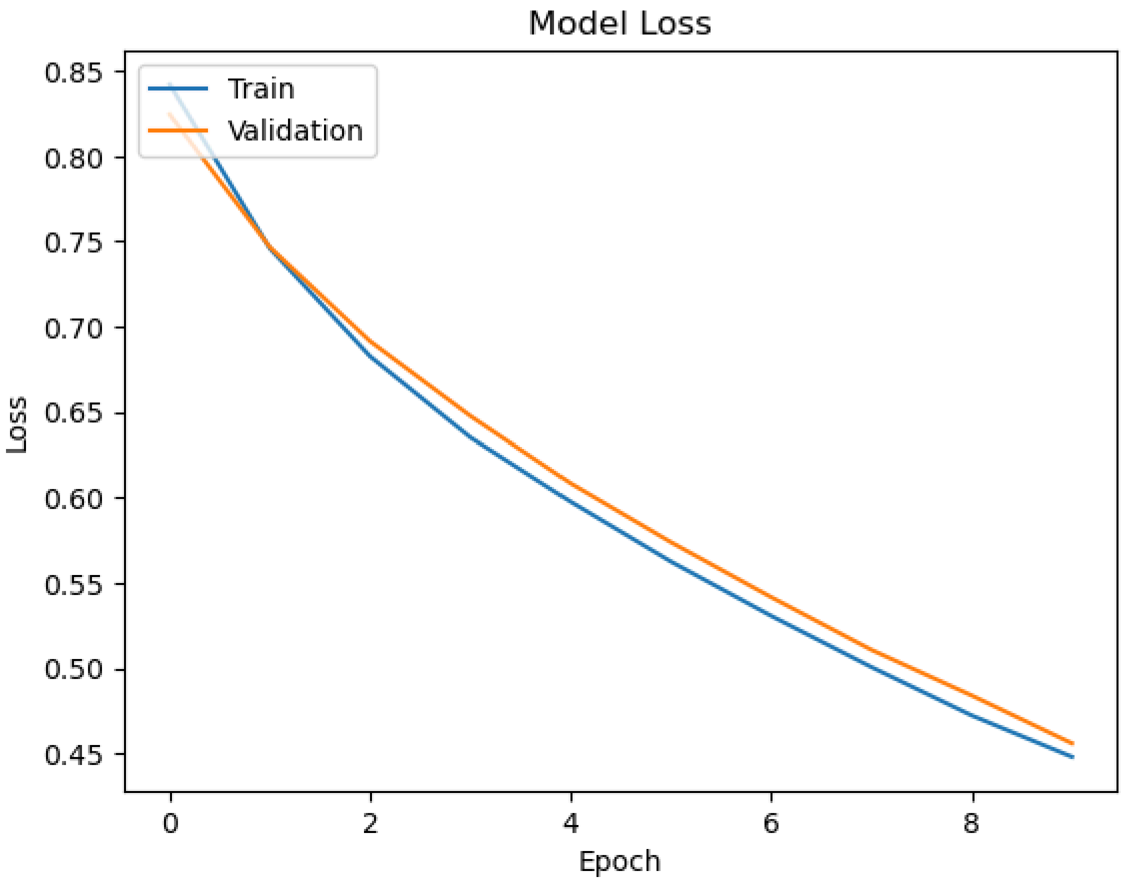
Insights for Graph Interpretations:

1. **Healthy Graph**: Training and validation lines rise together, with training slightly above validation, both stabilizing at similar values. The model learns effectively and generalizes well without significant overfitting or underfitting.

2. **Overfitting**: Training accuracy increases sharply, nearing 100%, while validation accuracy plateaus or declines, indicating the model is memorizing the training data. Address this with regularization, more data, or simplifying the model.

3. **Underfitting**: Both accuracies remain low and close together, showing the model is too simple or undertrained to capture patterns. Fix this by increasing model complexity, training longer, or reducing regularization.

In [42]:
```
# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Explanation for Loss vs. Epoch Graphs:

1. **Healthy Graph**: Training and validation loss decrease together, staying close. Indicates effective learning and good generalization without overfitting or underfitting.

2. **Overfitting**: Training loss drops steadily, but validation loss plateaus or increases. Suggests memorization of training data. Mitigate with regularization, early stopping, or simpler models.

3. **Underfitting**: Both losses remain high and decrease very slowly. Implies the model is too simple or inadequately trained. Address by adding complexity, training longer, or improving data preprocessing.

4. **Validation Loss Fluctuates**: Training loss decreases, but validation loss oscillates. May result from noise or insufficient validation data. Use larger datasets or techniques like data augmentation to stabilize.

Making Predictions with the Trained Model

Now that the model is trained, we can use it to make predictions on new, unseen data. Let's see how to predict the class labels for our test data.

Loss measures the "error" in predictions, guiding the model during training to improve its performance. It acts as the foundation for optimizing weights and ensuring the model learns meaningful patterns from the data.

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

- loss function is setup when the model is compiled
- `sparse_categorical_crossentropy` is used for multi-class classification problems.
- we can use various other loss functions too

In [41]:
```
test_loss, test_accuracy = model.evaluate(X, y)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 961us/step - accuracy: 0.8300 - loss: 0.4430
Test Loss: 0.4474693238735199
Test Accuracy: 0.8259999752044678
```

Loss Functions in Deep Learning

- **`sparse_categorical_crossentropy`** : For multi-class classification with integer labels; calculates the cross-entropy between true labels and predicted probabilities.
- **`categorical_crossentropy`** : For multi-class classification with one-hot encoded labels; computes cross-entropy for each class.
- **`binary_crossentropy`** : For binary classification; measures the log loss for two classes.
- **`mean_squared_error`** **(MSE)**: For regression; computes the squared difference between predicted and true values.
- **`mean_absolute_error`** **(MAE)**: For regression; computes the absolute difference between predictions and targets.
- **`huber_loss`** : For regression; combines MSE and MAE, robust to outliers.
- **`mean_absolute_percentage_error`** **(MAPE)**: For regression; calculates the percentage difference between predictions and actual values.
- **`hinge`** : For binary classification with labels -1 and 1; used in SVMs, penalizes misclassified samples.
- **`squared_hinge`** : Variation of hinge loss; penalizes the square of hinge loss for misclassifications.
- **`poisson`** : For count-based data; computes the Poisson deviance between true and predicted values.
- **`cosine_similarity`** : Measures the cosine similarity between true and predicted vectors; useful for directional data.
- **`log_cosh`** : For regression; similar to MSE but less sensitive to large errors.
- **`kullback_leibler_divergence`** **(KLD)**: Measures divergence between two probability distributions, true and predicted.
- **`custom_loss`** : User-defined function tailored to specific requirements in a model.

2. Create your own Neural Networks

Live Exercise

Let's create a **Neural Network** and apply it on the dataset we used earlier, to **Predict Customer Churn**

1. Start with importing libraries and dataset

In [1]:

```
# pip install pandas numpy tensorflow scikit-learn
```

In [2]:

```
# pip install kagglehub # if kagglehub is not installed yet
#import kagglehub

# Download latest version
#path = kagglehub.dataset_download("blastchar/telco-customer-churn")

#print("Path to dataset files:", path)
```

In [3]:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


# Load the dataset
# Use the path to get to the dataset
# Rename the dataset and choose the right path

# Load the dataset (assuming you have downloaded the dataset as a CSV)
data = pd.read_csv('c:\\Users\\devid\\.cache\\kagglehub\\datasets\\blastchar\\telco-customer-churn\\versions\\1\\Telco-Customer-Churn.csv')

# Display the first few rows
data.head()
```

Out[3]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	...	DeviceProtection
0	7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	...	No
1	5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	...	Yes
2	3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	...	No
3	7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	...	Yes
4	9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	...	No

5 rows × 21 columns

2. **Data Cleaning**

- We can check for missing values and handle them if needed
- We need to turn catagorical values into numeric

In [4]:

```
# Clean the dataset
# Drop unnecessary columns (e.g., customerID)
data = data.drop(columns=['customerID'])

# Handle missing or invalid values
# Convert 'TotalCharges' to numeric, replacing errors with NaN
data['TotalCharges'] = pd.to_numeric(data['TotalCharges'], errors='coerce')
data = data.dropna()  # Drop rows with NaN values
```

In [5]:

```
data.info() # check what kind of data there is
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 7032 entries, 0 to 7042
Data columns (total 20 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   gender            7032 non-null   object 
 1   SeniorCitizen     7032 non-null   int64  
 2   Partner           7032 non-null   object 
 3   Dependents        7032 non-null   object 
 4   tenure            7032 non-null   int64  
 5   PhoneService      7032 non-null   object 
 6   MultipleLines     7032 non-null   object 
 7   InternetService   7032 non-null   object 
 8   OnlineSecurity    7032 non-null   object 
 9   OnlineBackup      7032 non-null   object 
 10  DeviceProtection  7032 non-null   object 
 11  TechSupport       7032 non-null   object 
 12  StreamingTV       7032 non-null   object 
 13  StreamingMovies   7032 non-null   object 
 14  Contract          7032 non-null   object 
 15  PaperlessBilling  7032 non-null   object 
 16  PaymentMethod     7032 non-null   object 
 17  MonthlyCharges    7032 non-null   float64
 18  TotalCharges      7032 non-null   float64
 19  Churn             7032 non-null   object 
dtypes: float64(2), int64(2), object(16)
memory usage: 1.1+ MB
```

- **Encode Categorical Features**
- Categorical features need to be encoded into numerical values. You can use OneHotEncoder for this task:

In [6]:
```python
# Encode categorical features
categorical_columns = ['gender', 'Partner', 'Dependents', 'PhoneService',
                       'MultipleLines', 'InternetService', 'OnlineSecurity',
                       'DeviceProtection', 'TechSupport', 'StreamingTV',
                       'StreamingMovies', 'Contract', 'PaperlessBilling',
                       'PaymentMethod', 'Churn']
```

In [7]:
```python
# Apply label encoding to categorical columns
encoder = LabelEncoder()
for col in categorical_columns:
    data[col] = encoder.fit_transform(data[col])
```

Select the target feature

- standardize the numeric columns

In [8]:
```
# Separate features (X) and target (y)
X = data.drop(columns=['Churn'])  # Features
y = data['Churn']                 # Target (Churn)
```

In [9]:
```
# Standardize numeric columns
scaler = StandardScaler()
numeric_columns = ['tenure', 'MonthlyCharges', 'TotalCharges']
X[numeric_columns] = scaler.fit_transform(X[numeric_columns])
```

3. Train test split

In [10]:
```
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

4. Create a Neural Network:

A simple 3-layer neural network with two hidden layers:

- 16 neurons in the first hidden layer.
- 8 neurons in the second hidden layer.

Output layer uses the sigmoid activation function for binary classification.

In [11]:
```
# Build a simple neural network
model = Sequential([
    Dense(16, activation='relu', input_shape=(19,)),  # Input layer
    Dense(8, activation='relu'),                                 # Hidden layer
    Dense(1, activation='sigmoid')                              # Output layer (binary classification)
])
```

```
c:\Users\devid\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

You specified `input_shape=(19,)` because this is the number of features (columns) in your training data X_train

5. Training the model:

- Trained the model for 10 epochs with a batch size of 32.

In [12]:
```
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

In [13]:
```
# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[13], line 2
      1 # Train the model
----> 2 history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

File c:\Users\devid\anaconda3\Lib\site-packages\keras\src\utils\traceback_utils.py:122, in filter_traceback.<locals>.error_handler(*args, **kwargs)
    119     filtered_tb = _process_traceback_frames(e.__traceback__)
    120     # To get the full stack trace, call:
    121     # `keras.config.disable_traceback_filtering()`
--> 122     raise e.with_traceback(filtered_tb) from None
    123 finally:
    124     del filtered_tb

File c:\Users\devid\anaconda3\Lib\site-packages\optree\ops.py:752, in tree_map(func, tree, is_leaf, none_is_leaf, namespace, *rests)
    750 leaves, treespec = _C.flatten(tree, is_leaf, none_is_leaf, namespace)
    751 flat_args = [leaves] + [treespec.flatten_up_to(r) for r in rests]
--> 752 return treespec.unflatten(map(func, *flat_args))

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\generic.py:6643, in NDFrame.astype(self, dtype, copy, errors)
   6637     results = [
   6638         ser.astype(dtype, copy=copy, errors=errors) for _, ser in self.items()
   6639     ]
   6641 else:
   6642     # else, only a single dtype is given
-> 6643     new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=errors)
   6644     res = self._constructor_from_mgr(new_data, axes=new_data.axes)
   6645     return res.__finalize__(self, method="astype")

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\managers.py:430, in BaseBlockManager.astype(self, dtype, copy, errors)
    427 elif using_copy_on_write():
    428     copy = False
--> 430 return self.apply(
    431     "astype",
    432     dtype=dtype,
    433     copy=copy,
    434     errors=errors,
    435     using_cow=using_copy_on_write(),
    436 )

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\managers.py:363, in BaseBlockManager.apply(self, f, align_keys, **kwargs)
    361         applied = b.apply(f, **kwargs)
    362     else:
--> 363         applied = getattr(b, f)(**kwargs)
```

```
    364     result_blocks = extend_blocks(applied, result_blocks)
    366 out = type(self).from_blocks(result_blocks, self.axes)

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\blocks.py:758, in Block.astype(self, dtype, copy, errors, using_cow, s
queeze)
    755         raise ValueError("Can not squeeze with more than one column.")
    756     values = values[0, :]  # type: ignore[call-overload]
--> 758 new_values = astype_array_safe(values, dtype, copy=copy, errors=errors)
    760 new_values = maybe_coerce_values(new_values)
    762 refs = None

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:237, in astype_array_safe(values, dtype, copy, errors)
    234     dtype = dtype.numpy_dtype
    236 try:
--> 237     new_values = astype_array(values, dtype, copy=copy)
    238 except (ValueError, TypeError):
    239     # e.g. _astype_nansafe can fail on object-dtype of strings
    240     #  trying to convert to float
    241     if errors == "ignore":

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:182, in astype_array(values, dtype, copy)
    179     values = values.astype(dtype, copy=copy)
    181 else:
--> 182     values = _astype_nansafe(values, dtype, copy=copy)
    184 # in pandas we don't store numpy str dtypes, so convert to object
    185 if isinstance(dtype, np.dtype) and issubclass(values.dtype.type, str):

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:133, in _astype_nansafe(arr, dtype, copy, skipna)
    129     raise ValueError(msg)
    131 if copy or arr.dtype == object or dtype == object:
    132     # Explicit copy, or required since NumPy can't view from / to object.
--> 133     return arr.astype(dtype, copy=True)
    135 return arr.astype(dtype, copy=copy)

ValueError: could not convert string to float: 'No'
```

We got an error

Let's debug this issue:

In [14]:
```
print(X_train.dtypes)  # Check if any column in X_train is non-numeric
print(y_train.dtypes)  # Check if y_train is non-numeric
```

```
gender                int32
SeniorCitizen         int64
Partner               int32
Dependents            int32
tenure              float64
PhoneService          int32
MultipleLines         int32
InternetService       int32
OnlineSecurity        int32
OnlineBackup         object
DeviceProtection      int32
TechSupport           int32
StreamingTV           int32
StreamingMovies       int32
Contract              int32
PaperlessBilling      int32
PaymentMethod         int32
MonthlyCharges      float64
TotalCharges        float64
dtype: object
int32
```

If any column in X_train or the target y_train contains strings, it needs to be converted. But we can see that there are not strings.

It's clear that the column OnlineBackup is still an object, meaning it contains string values that need to be encoded to numerical values for the neural network to process.

In [15]:
```
# Encode 'OnlineBackup' column
encoder = LabelEncoder()
data['OnlineBackup'] = encoder.fit_transform(data['OnlineBackup'])
```

In [16]:
```
print(data.dtypes)  # All columns should now be numeric
```

```
gender                int32
SeniorCitizen         int64
Partner               int32
Dependents            int32
tenure                int64
PhoneService          int32
MultipleLines         int32
InternetService       int32
OnlineSecurity        int32
OnlineBackup          int32
DeviceProtection      int32
TechSupport           int32
StreamingTV           int32
StreamingMovies       int32
Contract              int32
PaperlessBilling      int32
PaymentMethod         int32
MonthlyCharges      float64
TotalCharges        float64
Churn                 int32
dtype: object
```

Proceed with training

In [17]:
```
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[17], line 1
----> 1 history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

File c:\Users\devid\anaconda3\Lib\site-packages\keras\src\utils\traceback_utils.py:122, in filter_traceback.<locals>.error_handler(*args, **
kwargs)
    119     filtered_tb = _process_traceback_frames(e.__traceback__)
    120     # To get the full stack trace, call:
    121     # `keras.config.disable_traceback_filtering()`
--> 122     raise e.with_traceback(filtered_tb) from None
    123 finally:
    124     del filtered_tb

File c:\Users\devid\anaconda3\Lib\site-packages\optree\ops.py:752, in tree_map(func, tree, is_leaf, none_is_leaf, namespace, *rests)
    750 leaves, treespec = _C.flatten(tree, is_leaf, none_is_leaf, namespace)
    751 flat_args = [leaves] + [treespec.flatten_up_to(r) for r in rests]
--> 752 return treespec.unflatten(map(func, *flat_args))

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\generic.py:6643, in NDFrame.astype(self, dtype, copy, errors)
   6637     results = [
   6638         ser.astype(dtype, copy=copy, errors=errors) for _, ser in self.items()
   6639     ]
   6641 else:
   6642     # else, only a single dtype is given
-> 6643     new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=errors)
   6644     res = self._constructor_from_mgr(new_data, axes=new_data.axes)
   6645     return res.__finalize__(self, method="astype")

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\managers.py:430, in BaseBlockManager.astype(self, dtype, copy, errors)
    427 elif using_copy_on_write():
    428     copy = False
--> 430 return self.apply(
    431     "astype",
    432     dtype=dtype,
    433     copy=copy,
    434     errors=errors,
    435     using_cow=using_copy_on_write(),
    436 )

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\managers.py:363, in BaseBlockManager.apply(self, f, align_keys, **kwar
gs)
    361         applied = b.apply(f, **kwargs)
    362     else:
--> 363         applied = getattr(b, f)(**kwargs)
    364     result_blocks = extend_blocks(applied, result_blocks)
```

```
    366 out = type(self).from_blocks(result_blocks, self.axes)

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\blocks.py:758, in Block.astype(self, dtype, copy, errors, using_cow, s
queeze)
    755         raise ValueError("Can not squeeze with more than one column.")
    756     values = values[0, :]  # type: ignore[call-overload]
--> 758 new_values = astype_array_safe(values, dtype, copy=copy, errors=errors)
    760 new_values = maybe_coerce_values(new_values)
    762 refs = None

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:237, in astype_array_safe(values, dtype, copy, errors)
    234     dtype = dtype.numpy_dtype
    236 try:
--> 237     new_values = astype_array(values, dtype, copy=copy)
    238 except (ValueError, TypeError):
    239     # e.g. _astype_nansafe can fail on object-dtype of strings
    240     #  trying to convert to float
    241     if errors == "ignore":

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:182, in astype_array(values, dtype, copy)
    179     values = values.astype(dtype, copy=copy)
    181 else:
--> 182     values = _astype_nansafe(values, dtype, copy=copy)
    184 # in pandas we don't store numpy str dtypes, so convert to object
    185 if isinstance(dtype, np.dtype) and issubclass(values.dtype.type, str):

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:133, in _astype_nansafe(arr, dtype, copy, skipna)
    129     raise ValueError(msg)
    131 if copy or arr.dtype == object or dtype == object:
    132     # Explicit copy, or required since NumPy can't view from / to object.
--> 133     return arr.astype(dtype, copy=True)
    135 return arr.astype(dtype, copy=copy)

ValueError: could not convert string to float: 'No'
```

In [18]:

```
# Check for non-numeric columns (any column that isn't of type 'number')
non_numeric_columns = X_train.select_dtypes(exclude=[np.number]).columns

print("Non-numeric columns:")
print(non_numeric_columns)

# Check if any non-numeric values exist in the dataset
non_numeric_rows = X_train[non_numeric_columns]
print("\nNon-numeric values:")
print(non_numeric_rows)
```

```
Non-numeric columns:
Index(['OnlineBackup'], dtype='object')

Non-numeric values:
             OnlineBackup
6030                   No
3410                   No
5483                   No
5524                   No
6337                   No
...                   ...
3778                   No
5199                  Yes
5235                   No
5399  No internet service
862                   Yes

[5625 rows x 1 columns]
```

It looks like the OnlineBackup column contains non-numeric values like 'No', 'Yes', and 'No internet service'. These need to be converted into numeric values for the neural network to process them.

In [19]:
```
X_train = X_train.drop('OnlineBackup', axis=1)
X_train = X_train.drop('InternetService', axis=1)
```

In [20]:
```
print(X_train.dtypes)  # All columns should now be numeric
```

```
gender                int32
SeniorCitizen         int64
Partner               int32
Dependents            int32
tenure              float64
PhoneService          int32
MultipleLines         int32
OnlineSecurity        int32
DeviceProtection      int32
TechSupport           int32
StreamingTV           int32
StreamingMovies       int32
Contract              int32
PaperlessBilling      int32
PaymentMethod         int32
MonthlyCharges      float64
TotalCharges        float64
dtype: object
```

```
In [21]: history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/10
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 1
----> 1 history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

File c:\Users\devid\anaconda3\Lib\site-packages\keras\src\utils\traceback_utils.py:122, in filter_traceback.<locals>.error_handler(*args, **
kwargs)
    119     filtered_tb = _process_traceback_frames(e.__traceback__)
    120     # To get the full stack trace, call:
    121     # `keras.config.disable_traceback_filtering()`
--> 122     raise e.with_traceback(filtered_tb) from None
    123 finally:
    124     del filtered_tb

File c:\Users\devid\anaconda3\Lib\site-packages\keras\src\layers\input_spec.py:227, in assert_input_compatibility(input_spec, inputs, layer_
name)
    222     for axis, value in spec.axes.items():
    223         if value is not None and shape[axis] not in {
    224             value,
    225             None,
    226         }:
--> 227             raise ValueError(
    228                 f'Input {input_index} of layer "{layer_name}" is '
    229                 f"incompatible with the layer: expected axis {axis} "
    230                 f"of input shape to have value {value}, "
    231                 "but received input with "
    232                 f"shape {shape}"
    233             )
    234 # Check shape.
    235 if spec.shape is not None:

ValueError: Exception encountered when calling Sequential.call().

Input 0 of layer "dense" is incompatible with the layer: expected axis -1 of input shape to have value 19, but received input with shape (No
ne, 17)

Arguments received by Sequential.call():
  • inputs=tf.Tensor(shape=(None, 17), dtype=float32)
  • training=True
  • mask=None
```

```
In [22]: print(X_train.shape)  # Check the number of columns (should be 18 now)
```

```
(5625, 17)
```

In [23]:
```python
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(17,)),  # Adjusted to 17 features
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
c:\Users\devid\anaconda3\Lib\site-packages\keras\src\layers\core\input_layer.py:26: UserWarning: Argument `input_shape` is deprecated. Use `
shape` instead.
  warnings.warn(
```

In [24]:
```python
print(X_train.dtypes)  # Check the data types of all columns
```

```
gender                int32
SeniorCitizen         int64
Partner               int32
Dependents            int32
tenure              float64
PhoneService          int32
MultipleLines         int32
OnlineSecurity        int32
DeviceProtection      int32
TechSupport           int32
StreamingTV           int32
StreamingMovies       int32
Contract              int32
PaperlessBilling      int32
PaymentMethod         int32
MonthlyCharges      float64
TotalCharges        float64
dtype: object
```

In [25]:
```python
print(X_train.isnull().sum())  # Check for missing values
```

```
gender              0
SeniorCitizen       0
Partner             0
Dependents          0
tenure              0
PhoneService        0
MultipleLines       0
OnlineSecurity      0
DeviceProtection    0
TechSupport         0
StreamingTV         0
StreamingMovies     0
Contract            0
PaperlessBilling    0
PaymentMethod       0
MonthlyCharges      0
TotalCharges        0
dtype: int64
```

In [26]:
```
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

In [27]:
```
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7256 - loss: 0.5373 - val_accuracy: 0.7876 - val_loss: 0.4233
Epoch 2/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7956 - loss: 0.4302 - val_accuracy: 0.7973 - val_loss: 0.4141
Epoch 3/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 899us/step - accuracy: 0.7994 - loss: 0.4185 - val_accuracy: 0.8000 - val_loss: 0.4101
Epoch 4/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 889us/step - accuracy: 0.8044 - loss: 0.4237 - val_accuracy: 0.8027 - val_loss: 0.4058
Epoch 5/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 915us/step - accuracy: 0.7993 - loss: 0.4242 - val_accuracy: 0.7991 - val_loss: 0.4092
Epoch 6/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 933us/step - accuracy: 0.8031 - loss: 0.4329 - val_accuracy: 0.8044 - val_loss: 0.4059
Epoch 7/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 909us/step - accuracy: 0.8010 - loss: 0.4244 - val_accuracy: 0.8053 - val_loss: 0.4076
Epoch 8/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 905us/step - accuracy: 0.8077 - loss: 0.4006 - val_accuracy: 0.8071 - val_loss: 0.4056
Epoch 9/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 915us/step - accuracy: 0.7850 - loss: 0.4392 - val_accuracy: 0.8044 - val_loss: 0.4098
Epoch 10/10
141/141 ━━━━━━━━━━━━━━━━━━━━ 0s 929us/step - accuracy: 0.8044 - loss: 0.4180 - val_accuracy: 0.8053 - val_loss: 0.4047
```

6. Evaluation:

- Let's monitor performance now.

In [28]:
```
test_loss, test_accuracy = model.evaluate(X_test, y_test, batch_size=32)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[28], line 1
----> 1 test_loss, test_accuracy = model.evaluate(X_test, y_test, batch_size=32)
      2 print(f"Test Loss: {test_loss}")
      3 print(f"Test Accuracy: {test_accuracy}")

File c:\Users\devid\anaconda3\Lib\site-packages\keras\src\utils\traceback_utils.py:122, in filter_traceback.<locals>.error_handler(*args, **kwargs)
    119     filtered_tb = _process_traceback_frames(e.__traceback__)
    120     # To get the full stack trace, call:
    121     # `keras.config.disable_traceback_filtering()`
--> 122     raise e.with_traceback(filtered_tb) from None
    123 finally:
    124     del filtered_tb

File c:\Users\devid\anaconda3\Lib\site-packages\optree\ops.py:752, in tree_map(func, tree, is_leaf, none_is_leaf, namespace, *rests)
    750 leaves, treespec = _C.flatten(tree, is_leaf, none_is_leaf, namespace)
    751 flat_args = [leaves] + [treespec.flatten_up_to(r) for r in rests]
--> 752 return treespec.unflatten(map(func, *flat_args))

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\generic.py:6643, in NDFrame.astype(self, dtype, copy, errors)
   6637     results = [
   6638         ser.astype(dtype, copy=copy, errors=errors) for _, ser in self.items()
   6639     ]
   6641 else:
   6642     # else, only a single dtype is given
-> 6643     new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=errors)
   6644     res = self._constructor_from_mgr(new_data, axes=new_data.axes)
   6645     return res.__finalize__(self, method="astype")

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\managers.py:430, in BaseBlockManager.astype(self, dtype, copy, errors)
    427 elif using_copy_on_write():
    428     copy = False
--> 430 return self.apply(
    431     "astype",
    432     dtype=dtype,
    433     copy=copy,
    434     errors=errors,
    435     using_cow=using_copy_on_write(),
    436 )

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\managers.py:363, in BaseBlockManager.apply(self, f, align_keys, **kwargs)
    361         applied = b.apply(f, **kwargs)
    362     else:
```

```
--> 363         applied = getattr(b, f)(**kwargs)
    364     result_blocks = extend_blocks(applied, result_blocks)
    366 out = type(self).from_blocks(result_blocks, self.axes)

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\internals\blocks.py:758, in Block.astype(self, dtype, copy, errors, using_cow, s
queeze)
    755         raise ValueError("Can not squeeze with more than one column.")
    756     values = values[0, :]  # type: ignore[call-overload]
--> 758 new_values = astype_array_safe(values, dtype, copy=copy, errors=errors)
    760 new_values = maybe_coerce_values(new_values)
    762 refs = None

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:237, in astype_array_safe(values, dtype, copy, errors)
    234     dtype = dtype.numpy_dtype
    236 try:
--> 237     new_values = astype_array(values, dtype, copy=copy)
    238 except (ValueError, TypeError):
    239     # e.g. _astype_nansafe can fail on object-dtype of strings
    240     #  trying to convert to float
    241     if errors == "ignore":

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:182, in astype_array(values, dtype, copy)
    179     values = values.astype(dtype, copy=copy)
    181 else:
--> 182     values = _astype_nansafe(values, dtype, copy=copy)
    184 # in pandas we don't store numpy str dtypes, so convert to object
    185 if isinstance(dtype, np.dtype) and issubclass(values.dtype.type, str):

File c:\Users\devid\anaconda3\Lib\site-packages\pandas\core\dtypes\astype.py:133, in _astype_nansafe(arr, dtype, copy, skipna)
    129     raise ValueError(msg)
    131 if copy or arr.dtype == object or dtype == object:
    132     # Explicit copy, or required since NumPy can't view from / to object.
--> 133     return arr.astype(dtype, copy=True)
    135 return arr.astype(dtype, copy=copy)

ValueError: could not convert string to float: 'No internet service'
```

There is another error, seems like the columns from earlier are persisting in test dataset too. Let's fix this:

In [30]:
```
X_test.dtypes
```

Out[30]:
```
gender                int32
SeniorCitizen         int64
Partner               int32
Dependents            int32
tenure              float64
PhoneService          int32
MultipleLines         int32
InternetService       int32
OnlineSecurity        int32
OnlineBackup         object
DeviceProtection      int32
TechSupport           int32
StreamingTV           int32
StreamingMovies       int32
Contract              int32
PaperlessBilling      int32
PaymentMethod         int32
MonthlyCharges      float64
TotalCharges        float64
dtype: object
```

In [31]:
```
# Drop multiple columns
X_test = X_test.drop(['InternetService', 'OnlineBackup'], axis=1)
```

In [32]:
```
# Check for accuracy

test_loss, test_accuracy = model.evaluate(X_test, y_test, batch_size=32)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
44/44 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.7678 - loss: 0.4312
Test Loss: 0.43959641456604004
Test Accuracy: 0.7768301367759705
```

Optional step: Make Predictions

In [33]:
```
# Make predictions (model outputs probabilities for class 1)
predictions = model.predict(X_test)

# Convert probabilities to binary predictions (0 or 1)
predictions = (predictions > 0.5).astype(int)
# This comparison checks if the predicted probability for class 1 is greater than 0.5. If it is, the output is True (1), else False (0).
# .astype(int): Converts the boolean True/False values into 1/0 for final binary classification.
```

```python
# If you want the predictions in a more readable format
print(predictions[:10])  # Print first 10 predictions
```

```
44/44 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
[[0]
 [0]
 [1]
 [0]
 [0]
 [1]
 [0]
 [1]
 [0]
 [0]]
```

Additional Resources:

- https://www.freecodecamp.org/news/building-a-neural-network-from-scratch
- Excellent introduction to Neural Networks by G. Sanderson: https://youtu.be/aircAruvnKk

__________________________END__________________________

THANK YOU!

Live Exercise Solutions

Programming Interveiw Questions

1. topic:

- question