Bank Customer Churn Prediction Using ANN

Problem Statement

- Make an Artificial Neural Network that can predict, based on geo-demographical and transactional information given above, if any individual customer will leave the bank or stay (customer churn). Besides, that rank all the customers of the bank, based on their probability of leaving. To do that, use the right Deep Learning model, one that is based on a probabilistic approach

Import libraries

In [1]:
```
# GPU : Graphical Processing Unit
# !pip install tenserflow
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow
print(tensorflow.__version__)
pd.options.display.max_rows = 500
```

```
2.9.2
```

In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

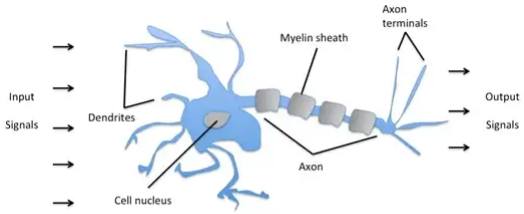
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Artificial Neural Network

- Artificial neural networks are one of the main tools used in machine learning. As the “neural” part of their name suggests, they are brain-inspired systems that are intended to replicate the way that we humans learn
- Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use
- They are excellent tools for finding patterns that are far too complex or numerous for a human programmer to extract and teach the machine to recognize
- While neural networks (also called “perceptrons”) have been around since the 1940s, it is only in the last several decades where they have become a major part of artificial intelligence. This is due to the arrival of a technique called “backpropagation,” which allows networks to adjust their hidden layers of neurons in situations where the outcome doesn’t match what the creator is hoping for — like a network designed to recognize dogs, which misidentifies a cat, for example
- Another important advance has been the arrival of deep learning neural networks, in which different layers of a multilayer network extract different features until it can recognize what it is looking for
- Source: https://medium.com/machine-learning-researcher/artificial-neural-network-ann-4481fa33d85a (https://medium.com/machine-learning-researcher/artificial-neural-network-ann-4481fa33d85a)

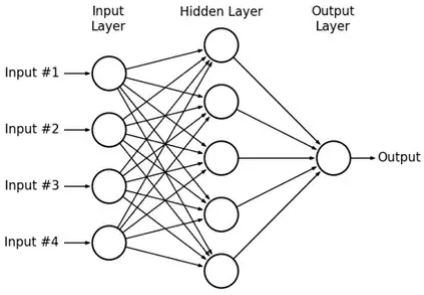
Basic Structure of ANNs

- The idea of ANNs is based on the belief that working of the human brain by making the right connections can be imitated using silicon and wires as living neurons and dendrites
- The human brain is composed of 86 billion nerve cells called neurons. They are connected to other thousand cells by Axons. Stimuli from the external environment or inputs from sensory organs are accepted by dendrites. These inputs create electric impulses, which quickly travel through the neural network. A neuron can then send the message to another neuron to handle the issue or does not send it forward



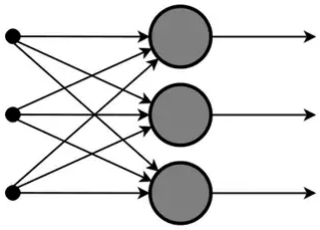
Schematic of a biological neuron.

- ANNs are composed of multiple nodes, which imitate biological neurons of the human brain. The neurons are connected by links and they interact with each other. The nodes can take input data and perform simple operations on the data. The result of these operations is passed to other neurons. The output at each node is called its activation or node value
- Each link is associated with weight. ANNs are capable of learning, which takes place by altering weight values

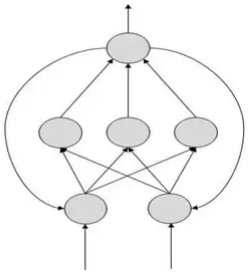


Types of Artificial Neural Networks

1. **FeedForward ANN**
    - In this ANN, the information flow is unidirectional. A unit sends information to another unit from which it does not receive any information. There are no feedback loops. They are used in pattern generation/recognition/classification. They have fixed inputs and outputs



1. **FeedBack ANN**
    - Here, feedback loops are allowed. They are used in content-addressable memories

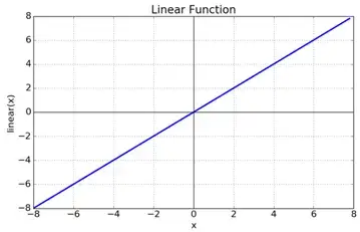


What is Activation Function?

- It's just a thing function that you use to get the output of the node. It is also known as Transfer Function
- Activation functions are really important for an Artificial Neural Network to learn and make sense of something reallocated and Non-linear complex functional mappings between the inputs and response variable. They introduce non-linear properties to our Network. Their main purpose is to convert an input signal of a node in an ANN to an output signal. That output signal now is used as an input in the next layer in the stack
- It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function)
- Specifically in A-NN we do the sum of products of inputs(X) and their corresponding Weights (W) and apply an Activation function f(x) to it to get the output of that layer and feed it as an input to the next layer
- Type of Activation Functions:
    1. Linear Activation Function As you can see the function is a line or linear. Therefore, the output of the functions will not be confined between any range.
    2. Non-linear Activation Functions

Type of Activation Functions:

1. **Linear Activation Function**
    - As you can see the function is a line or linear. Therefore, the output of the functions will not be confined between any range
    - Equation: f(x) = x
    - Range: (-infinity to infinity)
    - It doesn't help with the complexity of various parameters of usual data that is fed to the neural networks



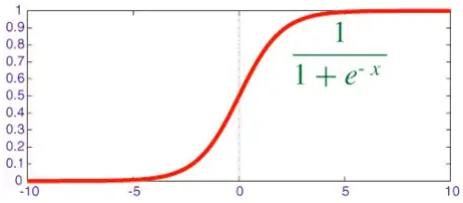
1. **Non-linear Activation Function**
    - The Nonlinear Activation Functions are the most used activation functions
    - It makes it easy for the model to generalize or adapt to a variety of data and to differentiate between the outputs

- The main terminologies needed to understand for nonlinear functions are:
    - **Derivative or Differential**: Change in y-axis w.r.t. change in the x-axis. It is also known as a slope
    - **Monotonic function**: A function that is either entirely non-increasing or non-decreasing

Different Types of Non-Linear Activation function

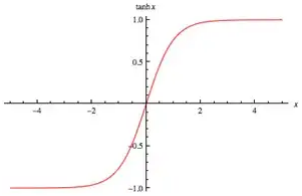
1. **Sigmoid Activation Function**

- The main reason why we use the sigmoid function is that it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since the probability of anything exists only between the range of 0 and 1, sigmoid is the right choice
- The function is differentiable. That means, we can find the slope of the sigmoid curve at any two points
- The function is monotonic but the function's derivative is not
- The logistic sigmoid function can cause a neural network to get stuck at the training time
- The **softmax function** is a more generalized logistic activation function that is used for multiclass classification



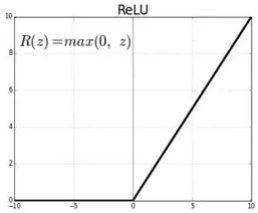
1. **Tanh Activation Function**

- tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s-shaped)
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph. The function is differentiable
- The function is monotonic while its derivative is not monotonic
- The tanh function is mainly used classification between two classes
- Both tanh and logistic sigmoid activation functions are used in feed-forward nets



1. **ReLU (Rectified Linear Unit) Activation Function**

- The ReLU is the most used activation function in the world right now. Since it is used in almost all the convolutional neural networks or deep learning
- As you will see in the below graph, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero
- Range: [0 to infinity)
- The function and it is derivative both are monotonic
- But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turn affects the resulting graph by not mapping the negative values appropriately



Note : Checkout the below link to study about back propgation in depth

- Source : https://medium.com/machine-learning-researcher/artificial-neural-network-ann-4481fa33d85a#:~:text=4%3A%20How%20Backpropagation,backpropagation%20algorithm%20work (https://medium.com/machine-learning-researcher/artificial-neural-network-ann-4481fa33d85a#:~:text=4%3A%20How%20Backpropagation,backpropagation%20algorithm%20work).

Import all the required libraries

In [2]:

Data Description

- RowNumber—corresponds to the record (row) number and has no effect on the output
- CustomerId—contains random values and has no effect on customer leaving the bank
- Surname—the surname of a customer has no impact on their decision to leave the bank
- CreditScore—can have an effect on customer churn, since a customer with a higher credit score is less likely to leave the bank
- Geography—a customer's location can affect their decision to leave the bank. Gender—it's interesting to explore whether gender plays a role in a customer leaving the bank
- Age—this is certainly relevant, since older customers are less likely to leave their bank than younger ones
- Tenure—refers to the number of years that the customer has been a client of the bank. Normally, older clients are more loyal and less likely to leave a bank
- Balance—also a very good indicator of customer churn, as people with a higher balance in their accounts are less likely to leave the bank compared to those with lower balances
- NumOfProducts—refers to the number of products that a customer has purchased through the bank
- HasCrCard—denotes whether or not a customer has a credit card. This column is also relevant, since people with a credit card are less likely to leave the bank
- IsActiveMember—active customers are less likely to leave the bank. EstimatedSalary—as with balance, people with lower salaries are more likely to leave the bank compared to those with higher salaries
- Exited—whether or not the customer left the bank

Objective of the problem

- It is advantageous for banks to know what leads a client towards the decision to leave the company
- Churn prevention allows companies to develop loyalty programs and retention campaigns to keep as many customers as possible

Data Ingestion

In [3]:
```
path = "/content/drive/MyDrive/FSDS_Job_Gurantee/Deep Learning/Bank_Customer_Churn_Prediction/churn.csv"
df = pd.read_csv(path)
```

In [4]:
```
df.isnull().sum()
```

Out[4]:
```
RowNumber          0
CustomerId         0
Surname            0
CreditScore        0
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

In [5]:
```
df.duplicated().sum()
```

Out[5]:
```
0
```

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   RowNumber        10000 non-null  int64  
 1   CustomerId       10000 non-null  int64  
 2   Surname          10000 non-null  object 
 3   CreditScore      10000 non-null  int64  
 4   Geography        10000 non-null  object 
 5   Gender           10000 non-null  object 
 6   Age              10000 non-null  int64  
 7   Tenure           10000 non-null  int64  
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64  
 10  HasCrCard        10000 non-null  int64  
 11  IsActiveMember   10000 non-null  int64  
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64  
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

In [7]:
```
df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
RowNumber	10000.0	5.000500e+03	2886.895680	1.00	2500.75	5.000500e+03	7.500250e+03	10000.00
CustomerId	10000.0	1.569094e+07	71936.186123	15565701.00	15628528.25	1.569074e+07	1.575323e+07	15815690.00
CreditScore	10000.0	6.505288e+02	96.653299	350.00	584.00	6.520000e+02	7.180000e+02	850.00
Age	10000.0	3.892180e+01	10.487806	18.00	32.00	3.700000e+01	4.400000e+01	92.00
Tenure	10000.0	5.012800e+00	2.892174	0.00	3.00	5.000000e+00	7.000000e+00	10.00
Balance	10000.0	7.648589e+04	62397.405202	0.00	0.00	9.719854e+04	1.276442e+05	250898.09
NumOfProducts	10000.0	1.530200e+00	0.581654	1.00	1.00	1.000000e+00	2.000000e+00	4.00
HasCrCard	10000.0	7.055000e-01	0.455840	0.00	0.00	1.000000e+00	1.000000e+00	1.00
IsActiveMember	10000.0	5.151000e-01	0.499797	0.00	0.00	1.000000e+00	1.000000e+00	1.00
EstimatedSalary	10000.0	1.000902e+05	57510.492818	11.58	51002.11	1.001939e+05	1.493882e+05	199992.48
Exited	10000.0	2.037000e-01	0.402769	0.00	0.00	0.000000e+00	0.000000e+00	1.00

Splitting data into independent and dependent features

In [8]:
```
X = df.iloc[ : , 3:13]
y = df.iloc[ : , -1]
```

Encoding the categorical data

In [9]:
```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

In [10]:
```
ohc = OneHotEncoder()
```

In [11]:
```
le_Geography = LabelEncoder()
X['Geography'] = le_Geography.fit_transform(X['Geography'])
le_Gender = LabelEncoder()
X['Gender'] = le_Gender.fit_transform(X['Gender']                                  )
```

Train test split

Note: Not performing in depth EDA because main goal of this notebook is to learn the practical implementation of ANN

In [12]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.2, random_state = 1)
```

Feature Scaling

In [13]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

ANN Model Building

In [14]:
```
# Importing the Keras libraries and packages
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
```

In [15]:
```
len(X_train[0])
```

Out[15]:
```
10
```

Initialize our ANN model

In [16]:
```
classifier = Sequential()
```

Adding the input layer and first hidden layer

- We use the Dense model to add a different layer. The parameter which we pass here first is units=10 neurons, which defines hidden layer=10, the second parameter is kernel_initializer ='he_uniform', basically this is a function that initializes the weights. The third parameter is activation= relu here in the first hidden layer we use relu activation. And the last parameter which we pass in dense function is input_dim= 10 which means the input node of our Neural Network is 10 because our dataset has 10 attributes that's why we choose 10 input nodes

In [17]:
```
# Adding Layers
# 'he_uniform' is wieght initilization techniques
# 10 dimension in input layer
# 10 unit in hidden layer, this is hyperparameter. As many as layer we can give
# 1 unit in output layer
classifier.add(Dense(units = 10, kernel_initializer = 'he_uniform', input_dim = 10)) #input layer
classifier.add(Dense(units = 10, kernel_initializer = 'he_uniform', activation = 'relu')) #hidden layer
```

Adding the output layer

- Add an output layer in our ANN structure units= 1 which means one output node here we use the sigmoid function because our target attribute has a binary class which is one or zero that's why we use sigmoid activation function

In [18]:
```
classifier.add(Dense(units = 1, kernel_initializer = 'glorot_uniform', activation = 'sigmoid')) # output layer,sigmoid because binary classification problem is there
```

Summary of the neural network

In [19]:
```
classifier.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                110

 dense_1 (Dense)             (None, 10)                110

 dense_2 (Dense)             (None, 1)                 11

=================================================================
Total params: 231
Trainable params: 231
Non-trainable params: 0
_________________________________________________________________
```

Compiling the ANN

- Compile the ANN to do that we use the compile method and add several parameters the first parameter is optimizer = Adam here use the optimal number of weights. So for choosing the optimal number of weights, there are various algorithms of Stochastic Gradient Descent but very efficient one which is Adam so that's why we use Adam optimizer here. The second parameter is loss this corresponds to loss function here we use binary_crossentropy because if we see target attribute our dataset which contains the binary value that's why we choose the binary cross-entropy. The final parameter is metrics basically It's a list of metrics to be evaluated by the model and here we choose the accuracy metrics

In [20]:
```
# Creating Back propgation
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

Fitting the ANN

- Fit the training data our model X_train, y_train is our training data. Here a batch size is basically a number of observations after which you want to update the weights here we take batch size 10. And the final parameter is epoch is basically when whole the training set passed through the ANN here we choose the 100 number of the epoch

In [21]:
```
model_history = classifier.fit(X_train, y_train, batch_size = 10, epochs = 30, validation_split = 0.25)
```

```
Epoch 1/30
600/600 [==============================] - 2s 3ms/step - loss: 0.5491 - accuracy: 0.7665 - val_loss: 0.4523 - val_accuracy: 0.8115
Epoch 2/30
600/600 [==============================] - 1s 2ms/step - loss: 0.4351 - accuracy: 0.8170 - val_loss: 0.4264 - val_accuracy: 0.8105
Epoch 3/30
600/600 [==============================] - 1s 2ms/step - loss: 0.4183 - accuracy: 0.8280 - val_loss: 0.4025 - val_accuracy: 0.8300
Epoch 4/30
600/600 [==============================] - 1s 2ms/step - loss: 0.4000 - accuracy: 0.8388 - val_loss: 0.3830 - val_accuracy: 0.8425
Epoch 5/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3853 - accuracy: 0.8457 - val_loss: 0.3649 - val_accuracy: 0.8540
Epoch 6/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3754 - accuracy: 0.8517 - val_loss: 0.3564 - val_accuracy: 0.8555
Epoch 7/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3691 - accuracy: 0.8535 - val_loss: 0.3558 - val_accuracy: 0.8535
Epoch 8/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3661 - accuracy: 0.8517 - val_loss: 0.3509 - val_accuracy: 0.8565
Epoch 9/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3643 - accuracy: 0.8537 - val_loss: 0.3458 - val_accuracy: 0.8540
Epoch 10/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3631 - accuracy: 0.8552 - val_loss: 0.3449 - val_accuracy: 0.8580
Epoch 11/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3614 - accuracy: 0.8562 - val_loss: 0.3441 - val_accuracy: 0.8530
Epoch 12/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3611 - accuracy: 0.8557 - val_loss: 0.3444 - val_accuracy: 0.8595
Epoch 13/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3607 - accuracy: 0.8562 - val_loss: 0.3451 - val_accuracy: 0.8585
Epoch 14/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3608 - accuracy: 0.8565 - val_loss: 0.3423 - val_accuracy: 0.8585
Epoch 15/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3600 - accuracy: 0.8562 - val_loss: 0.3392 - val_accuracy: 0.8565
Epoch 16/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3598 - accuracy: 0.8575 - val_loss: 0.3412 - val_accuracy: 0.8585
Epoch 17/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3590 - accuracy: 0.8568 - val_loss: 0.3401 - val_accuracy: 0.8570
Epoch 18/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3589 - accuracy: 0.8588 - val_loss: 0.3413 - val_accuracy: 0.8555
Epoch 19/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3585 - accuracy: 0.8563 - val_loss: 0.3427 - val_accuracy: 0.8580
Epoch 20/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3586 - accuracy: 0.8575 - val_loss: 0.3441 - val_accuracy: 0.8550
Epoch 21/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3582 - accuracy: 0.8540 - val_loss: 0.3406 - val_accuracy: 0.8545
Epoch 22/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3571 - accuracy: 0.8545 - val_loss: 0.3403 - val_accuracy: 0.8585
Epoch 23/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3578 - accuracy: 0.8558 - val_loss: 0.3401 - val_accuracy: 0.8570
Epoch 24/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3567 - accuracy: 0.8560 - val_loss: 0.3425 - val_accuracy: 0.8545
Epoch 25/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3569 - accuracy: 0.8550 - val_loss: 0.3396 - val_accuracy: 0.8570
Epoch 26/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3559 - accuracy: 0.8553 - val_loss: 0.3423 - val_accuracy: 0.8580
Epoch 27/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3566 - accuracy: 0.8592 - val_loss: 0.3443 - val_accuracy: 0.8575
Epoch 28/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3573 - accuracy: 0.8555 - val_loss: 0.3414 - val_accuracy: 0.8585
Epoch 29/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3570 - accuracy: 0.8518 - val_loss: 0.3389 - val_accuracy: 0.8550
Epoch 30/30
600/600 [==============================] - 1s 2ms/step - loss: 0.3565 - accuracy: 0.8567 - val_loss: 0.3396 - val_accuracy: 0.8560
```

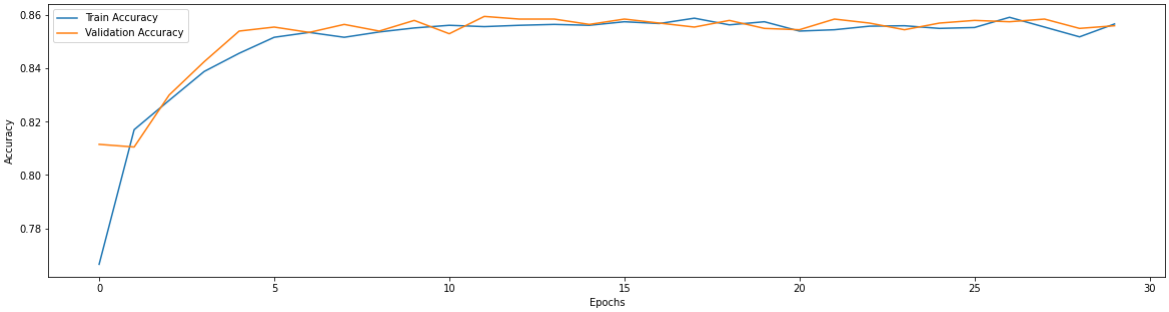
In [22]:
```
model_history.history['accuracy']
```

Out[22]:
```
[0.7664999961853027,
 0.8169999718666077,
 0.828000009059906,
 0.8388333320617676,
 0.8456666469573975,
 0.8516666889190674,
 0.8535000085830688,
 0.8516666889190674,
 0.8536666631698608,
 0.8551666736602783,
 0.856166660785675,
 0.8556666374206543,
 0.856166660785675,
 0.8565000295639038,
 0.856166660785675,
 0.8575000166893005,
 0.8568333387374878,
 0.8588333129882812,
 0.856333315372467,
 0.8575000166893005,
 0.8539999723434448,
 0.8544999957084656,
 0.8558333516120911,
 0.8560000061988831,
 0.8550000190734863,
 0.8553333282470703,
 0.85916668176651,
 0.8554999828338623,
 0.8518333435058594,
 0.8566666841506958]
```

Plot between train accuracy and validation accuracy

In [23]:
```
plt.figure(figsize = (20,5))
plt.plot(model_history.history['accuracy'], label = 'Train Accuracy')
plt.plot(model_history.history['val_accuracy'], label = 'Validation Accuracy')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
```

Out[23]:
```
Text(0, 0.5, 'Accuracy')
```



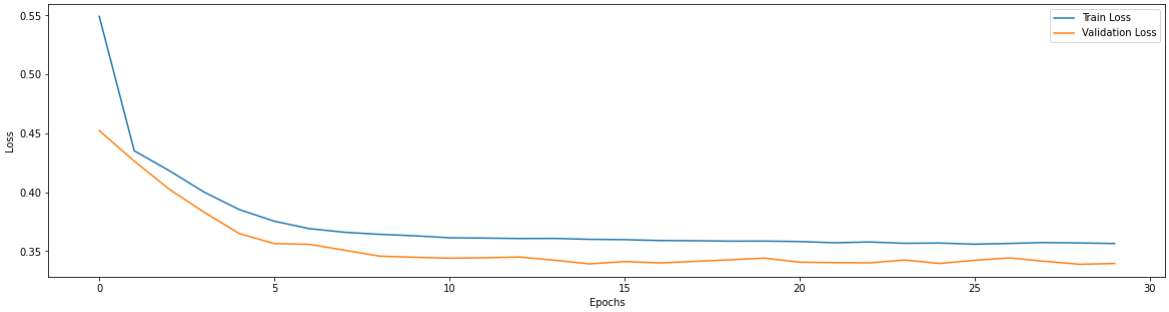
Observation:

- We can choose the optimal number of epochs based on the graph. The point where accuracy become saturate is the optimal number of epochs

Plot between train loss and validation loss

In [24]:
```
plt.figure(figsize = (20,5))
plt.plot(model_history.history['loss'], label = 'Train Loss')
plt.plot(model_history.history['val_loss'], label = 'Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

Out[24]:
```
Text(0, 0.5, 'Loss')
```



Observation:

- We can choose the optimal number of epochs based on the graph. The point where loss become saturate is the optimal number of epochs

Test Prediction

In [25]:
```
#probability
y_pred = classifier.predict(X_test)
```

```
63/63 [==============================] - 0s 1ms/step
```

In [26]:
```
# Thrsholf for the sigmoid is 0.5. Above 0.5, True and below that False
y_pred = np.where(y_pred > 0.5,1, 0)
```

Confusion Matrix

In [27]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_pred, y_test)
```

Out[27]:
```
array([[1536,  244],
       [  49,  171]])
```

Accuracy

Test Accuracy Prediction

In [28]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

Out[28]:
```
0.8535
```

Train Accuracy Prediction

In [29]:
```
#probability
y_pred_train = classifier.predict(X_train)
# Thrsholf for the sigmoid is 0.5. Above 0.5, True and below that False
y_pred_train = np.where(y_pred_train > 0.5,1, 0)
from sklearn.metrics import accuracy_score
accuracy_score(y_train, y_pred_train)
```

```
250/250 [==============================] - 0s 1ms/step
```

Out[29]:
```
0.857625
```

Conclusion:

- The purpose of this notebook was only for learning
- I have tried to comment on each part
- The accuracy of the model is 85 % and it is general model because train and test accuracy are close

END