ANN

February 18, 2025

[ ]:
```
'''
    Artificial Neural Network -->

    An Artificial Neural Network (ANN) is a computational model inspired by
    the structure and functioning of the human brain. It consists of layers
    of interconnected nodes (neurons) that process and learn from data.
    ANNs are a class of machine learning models that are capable of identifying
    patterns and making predictions based on data. They are widely used in
    various applications, such as image recognition, speech recognition,
    natural language processing, and more.
'''
```

[ ]:
```
'''
    Key Components -->

    Neurons (Nodes):

    These are the basic units of the network, similar to the neurons in the
    human brain. Each neuron receives input, processes it, and passes the
    result to other neurons.

    Layers:

    Input Layer : This is the first layer, where raw input data (such as pixel
    values in an image or features in a dataset) is fed into the network.
    Hidden Layers: These layers perform computations on the input data.
    ANNs can have multiple hidden layers, which allow the network to learn
    complex patterns and representations.
    Output Layer: The output layer produces the final prediction or␣
↪classification
    result based on the learned patterns.

    Weights:

    The connections between neurons are assigned weights, which determine the
    strength of the connections. During training, the weights are adjusted to
    minimize the error in predictions.
```

```
    Bias:

    A bias term is added to each neuron to help the network make better␣
↪predictions.
    It helps shift the activation function and allows the network to better fit
    the data.

    Activation Functions:

    After computing the weighted sum of inputs, an activation function is␣
↪applied
    to introduce non-linearity. Common activation functions include:

    ReLU (Rectified Linear Unit): Often used in hidden layers.
    Sigmoid: Used for binary classification, outputs values between 0 and 1.
    Softmax: Used for multi-class classification, outputs probability␣
↪distribution
    across multiple classes.

    Loss Function:

    A loss function measures the difference between the predicted output and the
    actual target. The goal during training is to minimize this loss.

    Optimization (Training):

    The process of adjusting the weights to minimize the loss function is done
    using optimization techniques like Gradient Descent or Adam. The network
    "learns" by updating the weights based on the errors.
'''
```

[ ]:
```
'''
    Working of an ANN -->

    Forward Propagation:
    During forward propagation, input data is passed through the layers of
    the network. At each layer, the input is transformed and passed to the
    next layer until the final prediction is made in the output layer.

    Backpropagation:
    Once the network produces an output, the error is calculated (using the
    loss function), and backpropagation is used to adjust the weights and
    biases in the network to reduce this error. This process is repeated
    iteratively to improve the network's accuracy.
'''
```

[ ]:
```
'''
    Intuition -->

    Imagine you're training a neural network to classify images as either
    "cat" or "dog." The network will:

    Receive pixel values of the image as input.
    Pass the data through hidden layers, where it learns patterns such as
    edges, shapes, and textures.
    The output layer will produce a probability indicating whether the image
    is more likely to be a cat or a dog.

    By adjusting the weights during training, the ANN gets better at
    classifying images over time.
'''
```

[37]:
```
#   Importing Libraries -->

import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
```

[2]:
```
tf.__version__
```

[2]: '2.18.0'

[3]:
```
#   Importing Dataset -->

data = pd.read_csv('Data/Churn_Modelling.csv')
data.head(10)
```

[3]:
```
   RowNumber  CustomerId   Surname  CreditScore Geography  Gender  Age  \
0          1    15634602  Hargrave          619    France  Female   42
1          2    15647311      Hill          608     Spain  Female   41
2          3    15619304      Onio          502    France  Female   42
3          4    15701354      Boni          699    France  Female   39
4          5    15737888  Mitchell          850     Spain  Female   43
5          6    15574012       Chu          645     Spain    Male   44
6          7    15592531  Bartlett          822    France    Male   50
```

```
7          8  15656148    Obinna          376   Germany  Female   29
8          9  15792365        He          501    France    Male   44
9         10  15592389        H?          684    France    Male   27

   Tenure    Balance  NumOfProducts  HasCrCard  IsActiveMember  \
0       2       0.00              1          1               1
1       1   83807.86              1          0               1
2       8  159660.80              3          1               0
3       1       0.00              2          0               0
4       2  125510.82              1          1               1
5       8  113755.78              2          1               0
6       7       0.00              2          1               1
7       4  115046.74              4          1               0
8       4  142051.07              2          0               1
9       2  134603.88              1          1               1

   EstimatedSalary  Exited
0        101348.88       1
1        112542.58       0
2        113931.57       1
3         93826.63       0
4         79084.10       0
5        149756.71       1
6         10062.80       0
7        119346.88       1
8         74940.50       0
9         71725.73       0
```

[8]:
```
#   Features and Target Seperation -->

x_data = data.iloc[:, 3:-1].values
y_data = data.iloc[:, -1].values
```

[9]:
```
x_data[:5]
```

[9]:
```
array([[619, 'France', 'Female', 42, 2, 0.0, 1, 1, 1, 101348.88],
       [608, 'Spain', 'Female', 41, 1, 83807.86, 1, 0, 1, 112542.58],
       [502, 'France', 'Female', 42, 8, 159660.8, 3, 1, 0, 113931.57],
       [699, 'France', 'Female', 39, 1, 0.0, 2, 0, 0, 93826.63],
       [850, 'Spain', 'Female', 43, 2, 125510.82, 1, 1, 1, 79084.1]],
      dtype=object)
```

[10]:
```
y_data[:5]
```

[10]:
```
array([1, 0, 1, 0, 0])
```

[ ]:
```python
#   Encoding Gender [Label Encoder] -->

encoder = LabelEncoder()
x_data[:, 2] = encoder.fit_transform(x_data[:, 2])
x_data[:5]
```

[ ]:
```
array([[619, 'France', 0, 42, 2, 0.0, 1, 1, 1, 101348.88],
       [608, 'Spain', 0, 41, 1, 83807.86, 1, 0, 1, 112542.58],
       [502, 'France', 0, 42, 8, 159660.8, 3, 1, 0, 113931.57],
       [699, 'France', 0, 39, 1, 0.0, 2, 0, 0, 93826.63],
       [850, 'Spain', 0, 43, 2, 125510.82, 1, 1, 1, 79084.1]],
      dtype=object)
```

[15]:
```python
#   Encoding Geography [One Hot Encoder] -->

col_transform = ColumnTransformer(transformers=[('encoder', OneHotEncoder(),␣
 ↪[1])], remainder='passthrough')
x_data = np.array(col_transform.fit_transform(x_data))
x_data[:5]
```

[15]:
```
array([[1.0, 0.0, 1.0, 0.0, 619, 0, 42, 2, 0.0, 1, 1, 1, 101348.88],
       [1.0, 0.0, 0.0, 1.0, 608, 0, 41, 1, 83807.86, 1, 0, 1, 112542.58],
       [1.0, 0.0, 1.0, 0.0, 502, 0, 42, 8, 159660.8, 3, 1, 0, 113931.57],
       [1.0, 0.0, 1.0, 0.0, 699, 0, 39, 1, 0.0, 2, 0, 0, 93826.63],
       [1.0, 0.0, 0.0, 1.0, 850, 0, 43, 2, 125510.82, 1, 1, 1, 79084.1]],
      dtype=object)
```

[17]:
```python
#   Splitting The Dataset -->

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
 ↪2, random_state=42)
```

[19]:
```python
#   Feature Scaling -->

sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

[20]:
```python
x_train[:5]
```

[20]:
```
array([[ 0.57946723, -0.57946723,  1.00150113, -0.57638802,  0.35649971,
         0.91324755, -0.6557859 ,  0.34567966, -1.21847056,  0.80843615,
         0.64920267,  0.97481699,  1.36766974],
       [-1.72572313,  1.72572313, -0.99850112, -0.57638802, -0.20389777,
         0.91324755,  0.29493847, -0.3483691 ,  0.69683765,  0.80843615,
         0.64920267,  0.97481699,  1.6612541 ],
       [ 0.57946723, -0.57946723, -0.99850112,  1.73494238, -0.96147213,
```

```
         0.91324755, -1.41636539, -0.69539349,  0.61862909, -0.91668767,
         0.64920267, -1.02583358, -0.25280688],
       [ 0.57946723, -0.57946723,  1.00150113, -0.57638802, -0.94071667,
        -1.09499335, -1.13114808,  1.38675281,  0.95321202, -0.91668767,
         0.64920267, -1.02583358,  0.91539272],
       [ 0.57946723, -0.57946723,  1.00150113, -0.57638802, -1.39733684,
         0.91324755,  1.62595257,  1.38675281,  1.05744869, -0.91668767,
        -1.54035103, -1.02583358, -1.05960019]])
```

[21]:
```
x_test[:5]
```

[21]:
```
array([[-1.72572313,  1.72572313, -0.99850112, -0.57638802, -0.57749609,
         0.91324755, -0.6557859 , -0.69539349,  0.32993735,  0.80843615,
        -1.54035103, -1.02583358, -1.01960511],
       [ 0.57946723, -0.57946723,  1.00150113, -0.57638802, -0.29729735,
         0.91324755,  0.3900109 , -1.38944225, -1.21847056,  0.80843615,
         0.64920267,  0.97481699,  0.79888291],
       [ 0.57946723, -0.57946723, -0.99850112,  1.73494238, -0.52560743,
        -1.09499335,  0.48508334, -0.3483691 , -1.21847056,  0.80843615,
         0.64920267, -1.02583358, -0.72797953],
       [-1.72572313,  1.72572313, -0.99850112, -0.57638802, -1.51149188,
         0.91324755,  1.91116988,  1.03972843,  0.68927246,  0.80843615,
         0.64920267,  0.97481699,  1.22138664],
       [ 0.57946723, -0.57946723, -0.99850112,  1.73494238, -0.9510944 ,
        -1.09499335, -1.13114808,  0.69270405,  0.78283876, -0.91668767,
         0.64920267,  0.97481699,  0.24756011]])
```

[28]:
```
#   Building The ANN -->

model = tf.keras.Sequential([
    keras.layers.Dense(6, activation='relu'),
    keras.layers.Dense(6, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])
```

[29]:
```
#   Compiling The ANN -->

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

[33]:
```
#   Summarizing The Model -->

model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(32, 6)	84
dense_4 (Dense)	(32, 6)	42
dense_5 (Dense)	(32, 1)	7

Total params: 401 (1.57 KB)

Trainable params: 133 (532.00 B)

Non-trainable params: 0 (0.00 B)

Optimizer params: 268 (1.05 KB)

[30]:

```
#   Training The ANN -->

model.fit(
    x_train,
    y_train,
    batch_size=32,
    epochs=100
)
```

```
Epoch 1/100
250/250              1s 1ms/step -
accuracy: 0.6747 - loss: 0.6632
Epoch 2/100
250/250              0s 1ms/step -
accuracy: 0.7937 - loss: 0.5070
Epoch 3/100
250/250              0s 1ms/step -
accuracy: 0.8067 - loss: 0.4536
Epoch 4/100
250/250              0s 1ms/step -
accuracy: 0.8011 - loss: 0.4524
Epoch 5/100
250/250              0s 2ms/step -
accuracy: 0.8168 - loss: 0.4283
Epoch 6/100
250/250              0s 2ms/step -
accuracy: 0.8191 - loss: 0.4180
Epoch 7/100
```

```
250/250               0s 2ms/step -
accuracy: 0.8231 - loss: 0.4073
Epoch 8/100
250/250               0s 2ms/step -
accuracy: 0.8266 - loss: 0.4004
Epoch 9/100
250/250               0s 2ms/step -
accuracy: 0.8371 - loss: 0.3803
Epoch 10/100
250/250               0s 2ms/step -
accuracy: 0.8471 - loss: 0.3747
Epoch 11/100
250/250               0s 1ms/step -
accuracy: 0.8470 - loss: 0.3722
Epoch 12/100
250/250               0s 1ms/step -
accuracy: 0.8479 - loss: 0.3630
Epoch 13/100
250/250               0s 1ms/step -
accuracy: 0.8480 - loss: 0.3679
Epoch 14/100
250/250               0s 1ms/step -
accuracy: 0.8503 - loss: 0.3586
Epoch 15/100
250/250               0s 1ms/step -
accuracy: 0.8556 - loss: 0.3534
Epoch 16/100
250/250               0s 1ms/step -
accuracy: 0.8560 - loss: 0.3504
Epoch 17/100
250/250               0s 1ms/step -
accuracy: 0.8608 - loss: 0.3388
Epoch 18/100
250/250               0s 1ms/step -
accuracy: 0.8544 - loss: 0.3486
Epoch 19/100
250/250               0s 1ms/step -
accuracy: 0.8546 - loss: 0.3486
Epoch 20/100
250/250               0s 1ms/step -
accuracy: 0.8547 - loss: 0.3410
Epoch 21/100
250/250               0s 2ms/step -
accuracy: 0.8602 - loss: 0.3426
Epoch 22/100
250/250               0s 2ms/step -
accuracy: 0.8675 - loss: 0.3339
Epoch 23/100
```

```
250/250             0s 1ms/step -
accuracy: 0.8581 - loss: 0.3462
Epoch 24/100
250/250             0s 1ms/step -
accuracy: 0.8581 - loss: 0.3470
Epoch 25/100
250/250             0s 1ms/step -
accuracy: 0.8600 - loss: 0.3509
Epoch 26/100
250/250             0s 1ms/step -
accuracy: 0.8606 - loss: 0.3494
Epoch 27/100
250/250             0s 1ms/step -
accuracy: 0.8609 - loss: 0.3406
Epoch 28/100
250/250             0s 1ms/step -
accuracy: 0.8605 - loss: 0.3354
Epoch 29/100
250/250             0s 1ms/step -
accuracy: 0.8611 - loss: 0.3404
Epoch 30/100
250/250             0s 1ms/step -
accuracy: 0.8610 - loss: 0.3454
Epoch 31/100
250/250             0s 1ms/step -
accuracy: 0.8688 - loss: 0.3345
Epoch 32/100
250/250             0s 2ms/step -
accuracy: 0.8632 - loss: 0.3368
Epoch 33/100
250/250             0s 2ms/step -
accuracy: 0.8560 - loss: 0.3476
Epoch 34/100
250/250             0s 1ms/step -
accuracy: 0.8576 - loss: 0.3425
Epoch 35/100
250/250             0s 1ms/step -
accuracy: 0.8692 - loss: 0.3317
Epoch 36/100
250/250             0s 1ms/step -
accuracy: 0.8637 - loss: 0.3359
Epoch 37/100
250/250             0s 1ms/step -
accuracy: 0.8540 - loss: 0.3515
Epoch 38/100
250/250             1s 2ms/step -
accuracy: 0.8608 - loss: 0.3350
Epoch 39/100
```

```
250/250              0s 2ms/step -
accuracy: 0.8677 - loss: 0.3342
Epoch 40/100
250/250              0s 2ms/step -
accuracy: 0.8625 - loss: 0.3368
Epoch 41/100
250/250              0s 1ms/step -
accuracy: 0.8683 - loss: 0.3305
Epoch 42/100
250/250              0s 1ms/step -
accuracy: 0.8565 - loss: 0.3419
Epoch 43/100
250/250              0s 1ms/step -
accuracy: 0.8624 - loss: 0.3397
Epoch 44/100
250/250              0s 1ms/step -
accuracy: 0.8625 - loss: 0.3371
Epoch 45/100
250/250              0s 1ms/step -
accuracy: 0.8617 - loss: 0.3396
Epoch 46/100
250/250              0s 2ms/step -
accuracy: 0.8637 - loss: 0.3369
Epoch 47/100
250/250              0s 1ms/step -
accuracy: 0.8628 - loss: 0.3393
Epoch 48/100
250/250              0s 1ms/step -
accuracy: 0.8604 - loss: 0.3384
Epoch 49/100
250/250              0s 1ms/step -
accuracy: 0.8591 - loss: 0.3444
Epoch 50/100
250/250              0s 2ms/step -
accuracy: 0.8654 - loss: 0.3286
Epoch 51/100
250/250              0s 2ms/step -
accuracy: 0.8547 - loss: 0.3501
Epoch 52/100
250/250              0s 1ms/step -
accuracy: 0.8616 - loss: 0.3408
Epoch 53/100
250/250              0s 1ms/step -
accuracy: 0.8673 - loss: 0.3262
Epoch 54/100
250/250              0s 1ms/step -
accuracy: 0.8592 - loss: 0.3419
Epoch 55/100
```

```
250/250              0s 1ms/step -
accuracy: 0.8618 - loss: 0.3409
Epoch 56/100
250/250              0s 2ms/step -
accuracy: 0.8600 - loss: 0.3373
Epoch 57/100
250/250              0s 2ms/step -
accuracy: 0.8601 - loss: 0.3318
Epoch 58/100
250/250              0s 1ms/step -
accuracy: 0.8546 - loss: 0.3456
Epoch 59/100
250/250              0s 1ms/step -
accuracy: 0.8617 - loss: 0.3442
Epoch 60/100
250/250              0s 1ms/step -
accuracy: 0.8621 - loss: 0.3329
Epoch 61/100
250/250              0s 1ms/step -
accuracy: 0.8548 - loss: 0.3474
Epoch 62/100
250/250              0s 2ms/step -
accuracy: 0.8660 - loss: 0.3262
Epoch 63/100
250/250              0s 2ms/step -
accuracy: 0.8622 - loss: 0.3436
Epoch 64/100
250/250              0s 2ms/step -
accuracy: 0.8611 - loss: 0.3363
Epoch 65/100
250/250              0s 1ms/step -
accuracy: 0.8679 - loss: 0.3263
Epoch 66/100
250/250              0s 2ms/step -
accuracy: 0.8654 - loss: 0.3296
Epoch 67/100
250/250              0s 2ms/step -
accuracy: 0.8524 - loss: 0.3460
Epoch 68/100
250/250              0s 2ms/step -
accuracy: 0.8694 - loss: 0.3236
Epoch 69/100
250/250              0s 1ms/step -
accuracy: 0.8637 - loss: 0.3271
Epoch 70/100
250/250              0s 1ms/step -
accuracy: 0.8604 - loss: 0.3367
Epoch 71/100
```

```
250/250                0s 1ms/step -
accuracy: 0.8638 - loss: 0.3397
Epoch 72/100
250/250                0s 1ms/step -
accuracy: 0.8594 - loss: 0.3482
Epoch 73/100
250/250                0s 1ms/step -
accuracy: 0.8695 - loss: 0.3254
Epoch 74/100
250/250                0s 1ms/step -
accuracy: 0.8608 - loss: 0.3420
Epoch 75/100
250/250                0s 1ms/step -
accuracy: 0.8671 - loss: 0.3296
Epoch 76/100
250/250                0s 2ms/step -
accuracy: 0.8678 - loss: 0.3247
Epoch 77/100
250/250                0s 1ms/step -
accuracy: 0.8616 - loss: 0.3371
Epoch 78/100
250/250                0s 1ms/step -
accuracy: 0.8608 - loss: 0.3375
Epoch 79/100
250/250                0s 2ms/step -
accuracy: 0.8669 - loss: 0.3315
Epoch 80/100
250/250                0s 1ms/step -
accuracy: 0.8589 - loss: 0.3369
Epoch 81/100
250/250                0s 1ms/step -
accuracy: 0.8733 - loss: 0.3198
Epoch 82/100
250/250                0s 1ms/step -
accuracy: 0.8571 - loss: 0.3442
Epoch 83/100
250/250                1s 2ms/step -
accuracy: 0.8674 - loss: 0.3257
Epoch 84/100
250/250                1s 2ms/step -
accuracy: 0.8559 - loss: 0.3434
Epoch 85/100
250/250                0s 1ms/step -
accuracy: 0.8677 - loss: 0.3287
Epoch 86/100
250/250                0s 1ms/step -
accuracy: 0.8588 - loss: 0.3439
Epoch 87/100
```

```
250/250              0s 2ms/step -
accuracy: 0.8638 - loss: 0.3339
Epoch 88/100
250/250              0s 2ms/step -
accuracy: 0.8698 - loss: 0.3285
Epoch 89/100
250/250              0s 2ms/step -
accuracy: 0.8578 - loss: 0.3360
Epoch 90/100
250/250              0s 2ms/step -
accuracy: 0.8694 - loss: 0.3221
Epoch 91/100
250/250              0s 2ms/step -
accuracy: 0.8675 - loss: 0.3253
Epoch 92/100
250/250              0s 1ms/step -
accuracy: 0.8663 - loss: 0.3331
Epoch 93/100
250/250              0s 1ms/step -
accuracy: 0.8653 - loss: 0.3323
Epoch 94/100
250/250              0s 2ms/step -
accuracy: 0.8669 - loss: 0.3318
Epoch 95/100
250/250              0s 2ms/step -
accuracy: 0.8632 - loss: 0.3335
Epoch 96/100
250/250              0s 2ms/step -
accuracy: 0.8635 - loss: 0.3344
Epoch 97/100
250/250              0s 1ms/step -
accuracy: 0.8639 - loss: 0.3320
Epoch 98/100
250/250              0s 1ms/step -
accuracy: 0.8657 - loss: 0.3243
Epoch 99/100
250/250              0s 1ms/step -
accuracy: 0.8679 - loss: 0.3296
Epoch 100/100
250/250              0s 1ms/step -
accuracy: 0.8575 - loss: 0.3463
```

[30]:
```
<keras.src.callbacks.history.History at 0x1bfc0f83790>
```

[39]:
```
#   Saving Model -->

model.save(filepath='./model.keras')
```

[36]:
```python
#   Predicting Test Results -->

y_pred = model.predict(x_test)
y_pred = (y_pred > 0.5)
y_pred[:20]
```

```
63/63            0s 1ms/step
```

[36]:
```
array([[False],
       [False],
       [False],
       [False],
       [False],
       [False],
       [False],
       [False],
       [False],
       [ True],
       [ True],
       [ True],
       [ True],
       [False],
       [False],
       [False],
       [False],
       [False],
       [False],
       [False]])
```

[41]:
```python
#   Metrics -->

acc_score = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
```

[42]:
```python
print("Accuracy Score --> ", acc_score)
```

```
Accuracy Score -->  0.8595
```

[43]:
```python
print("Confusion Matrix -->\n\n", conf_matrix)
```

```
Confusion Matrix -->

 [[1538   69]
 [ 212  181]]
```

[44]:
```python
print("Classification Report -->\n\n", class_report)
```

```
Classification Report -->
```

	precision	recall	f1-score	support
0	0.88	0.96	0.92	1607
1	0.72	0.46	0.56	393
accuracy			0.86	2000
macro avg	0.80	0.71	0.74	2000
weighted avg	0.85	0.86	0.85	2000