artificial_neural_network

June 21, 2021

1 Artificial Neural Network (ANN)

Work by: Raghavendra Tapas

1.0.1 Importing the libraries

[1]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
```

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

[2]:
```
'2.5.0'
```

1.1 Data Preprocessing

1.1.1 Importing the dataset

- x - all the features except exited column
- y - value to be predicted

[22]:
```
dataset = pd.read_csv('Churn_Modelling.csv')
x = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values
```

[23]:
```
dataset
```

[23]:
```
      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
…           …          …          …          …         …       …   …
9995       9996    15606229   Obijiaku          771    France    Male   39
9996       9997    15569892  Johnstone          516    France    Male   35
9997       9998    15584532        Liu          709    France  Female   36
9998       9999    15682355  Sabbatini          772   Germany    Male   42
9999      10000    15628319     Walker          792    France  Female   28
```

```
      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
…        …         …             …         …              …
9995       5       0.00              2          1               0
9996      10   57369.61              1          1               1
9997       7       0.00              1          0               1
9998       3   75075.31              2          1               0
9999       4  130142.79              1          1               0

      EstimatedSalary  Exited
0           101348.88       1
1           112542.58       0
2           113931.57       1
3            93826.63       0
4            79084.10       0
…                 …     …
9995         96270.64       0
9996        101699.77       0
9997         42085.58       1
9998         92888.52       1
9999         38190.78       0

[10000 rows x 14 columns]
```

[24]:
```python
print(x)
```

```
[[619 'France' 'Female' … 1 1 101348.88]
 [608 'Spain' 'Female' … 0 1 112542.58]
 [502 'France' 'Female' … 1 0 113931.57]
 …
 [709 'France' 'Female' … 0 1 42085.58]
 [772 'Germany' 'Male' … 1 0 92888.52]
 [792 'France' 'Female' … 1 0 38190.78]]
```

[25]:
```python
print(y)
```

```
[1 0 1 … 1 1 0]
```

1.1.2 Encoding Categorical Data

Label Encoding the "Gender" column

```
[26]: from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
x[:, 2] = le.fit_transform(x[:, 2])
```

```
[27]: type(x)
```

```
[27]: numpy.ndarray
```

One Hot Encoding the "Geography" column

```
[8]: from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers = [('encoder', OneHotEncoder(), [1])],␣
 ↪remainder='passthrough')
x = np.array(ct.fit_transform(x))
```

1.1.3 Splitting the dataset into the Training set and Test set

```
[9]: 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 = 0)
```

1.1.4 Feature Scaling

```
[10]: from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

1.2 Building the ANN

How many neurons to choose is a trial and error process in deep learning.

- **1st layer:** `rectifier` activation function, `6` neurons
- **2nd layer:** `rectifier` activation function, `6` neurons
- **Output layer:** `sigmoid` activation function, `1` neuron, because the variable we are trying to predict is `binary classification` problem.

i.e. Whether or not the customer leaves the bank. (Yes or No)

1.2.1 Initializing the ANN

```
[28]: ann = tf.keras.models.Sequential()
```

1.2.2 Adding the input layer and the first hidden layer

```
[29]: ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

1.2.3 Adding the second hidden layer

[30]:
```
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

1.2.4 Adding the output layer

[31]:
```
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

1.3 Training the ANN

- `Adam optimizer`: Adaptive Moment Estimation or Adam optimizer combines two gradient descent methodologies. Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters.
- `Binary Cross-Entropy`: Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events. Since we're trying to compute a loss, we need to penalize bad predictions. If the probability associated with the true class is 1.0, we need its loss to be zero. Conversely, if that probability is low, say, 0.01, we need its loss to be huge.

1.3.1 Compiling the ANN

[32]:
```
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =␣
↪['accuracy'])
```

1.3.2 Training the ANN on the Training set

- Batch gradient descent is faster to compute. Usually batch size is taken to be 32.
- One epoch consists of one full training cycle on the training set. Once every sample in the set is seen, you start again - marking the beginning of the 2nd epoch.

[34]:
```
ann.fit(x_train, y_train, batch_size = 32, epochs = 100)
```

```
Epoch 1/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3358 -
accuracy: 0.8650
Epoch 2/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3339 -
accuracy: 0.8644
Epoch 3/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3332 -
accuracy: 0.8671
Epoch 4/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3335 -
accuracy: 0.8636
Epoch 5/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3327 -
accuracy: 0.8671
```

```
Epoch 6/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3327 -
accuracy: 0.8670
Epoch 7/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3319 -
accuracy: 0.8666
Epoch 8/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3325 -
accuracy: 0.8643
Epoch 9/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3323 -
accuracy: 0.8639
Epoch 10/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3317 -
accuracy: 0.8668
Epoch 11/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3321 -
accuracy: 0.8648
Epoch 12/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3317 -
accuracy: 0.8649
Epoch 13/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3319 -
accuracy: 0.8629
Epoch 14/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3309 -
accuracy: 0.8645
Epoch 15/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3310 -
accuracy: 0.8677
Epoch 16/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3317 -
accuracy: 0.8664
Epoch 17/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3309 -
accuracy: 0.8649
Epoch 18/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3314 -
accuracy: 0.8656
Epoch 19/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3314 -
accuracy: 0.8665
Epoch 20/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3307 -
accuracy: 0.8640
Epoch 21/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3308 -
accuracy: 0.8662
```

```
Epoch 22/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3304 -
accuracy: 0.8654
Epoch 23/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3312 -
accuracy: 0.8643
Epoch 24/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3306 -
accuracy: 0.8656
Epoch 25/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3306 -
accuracy: 0.8652
Epoch 26/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3301 -
accuracy: 0.8654
Epoch 27/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3305 -
accuracy: 0.8655
Epoch 28/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3304 -
accuracy: 0.8652
Epoch 29/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3306 -
accuracy: 0.8635
Epoch 30/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3298 -
accuracy: 0.8662
Epoch 31/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3301 -
accuracy: 0.8649
Epoch 32/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3292 -
accuracy: 0.8652
Epoch 33/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3299 -
accuracy: 0.8654
Epoch 34/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3293 -
accuracy: 0.8652
Epoch 35/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3301 -
accuracy: 0.8675
Epoch 36/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3295 -
accuracy: 0.8654
Epoch 37/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3302 -
accuracy: 0.8640
```

```
Epoch 38/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3298 -
accuracy: 0.8645
Epoch 39/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3296 -
accuracy: 0.8627
Epoch 40/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3298 -
accuracy: 0.8643
Epoch 41/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3296 -
accuracy: 0.8640
Epoch 42/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3299 -
accuracy: 0.8649
Epoch 43/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3298 -
accuracy: 0.8675
Epoch 44/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3300 -
accuracy: 0.8639
Epoch 45/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3286 -
accuracy: 0.8650
Epoch 46/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3294 -
accuracy: 0.8659
Epoch 47/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3285 -
accuracy: 0.8652
Epoch 48/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3289 -
accuracy: 0.8659
Epoch 49/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3294 -
accuracy: 0.8650
Epoch 50/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3293 -
accuracy: 0.8640
Epoch 51/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3289 -
accuracy: 0.8643
Epoch 52/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3289 -
accuracy: 0.8644: 0s - loss:
Epoch 53/100
667/667 [==============================] - ETA: 0s - loss: 0.3291 - accuracy:
0.8649 ETA: 0s - loss: 0.331 - 2s 3ms/step - loss: 0.3289 - accuracy: 0.8648
```

```
Epoch 54/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3288 -
accuracy: 0.8664
Epoch 55/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3291 -
accuracy: 0.8656
Epoch 56/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3290 -
accuracy: 0.8650
Epoch 57/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3283 -
accuracy: 0.8650
Epoch 58/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3290 -
accuracy: 0.8652
Epoch 59/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3285 -
accuracy: 0.8658
Epoch 60/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3283 -
accuracy: 0.8650
Epoch 61/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3287 -
accuracy: 0.8654
Epoch 62/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3282 -
accuracy: 0.8661
Epoch 63/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3283 -
accuracy: 0.8659
Epoch 64/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3285 -
accuracy: 0.8652
Epoch 65/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3283 -
accuracy: 0.8650
Epoch 66/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3279 -
accuracy: 0.8671
Epoch 67/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3280 -
accuracy: 0.8630
Epoch 68/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3284 -
accuracy: 0.8644
Epoch 69/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3279 -
accuracy: 0.8675
```

```
Epoch 70/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3283 -
accuracy: 0.8648
Epoch 71/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3272 -
accuracy: 0.8649
Epoch 72/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3281 -
accuracy: 0.8656
Epoch 73/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3277 -
accuracy: 0.8651
Epoch 74/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3279 -
accuracy: 0.8668
Epoch 75/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3287 -
accuracy: 0.8636
Epoch 76/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3280 -
accuracy: 0.8658
Epoch 77/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3279 -
accuracy: 0.8656
Epoch 78/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3279 -
accuracy: 0.8656
Epoch 79/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3278 -
accuracy: 0.8633
Epoch 80/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3277 -
accuracy: 0.8680
Epoch 81/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3277 -
accuracy: 0.8683
Epoch 82/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3274 -
accuracy: 0.8664
Epoch 83/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3280 -
accuracy: 0.8666
Epoch 84/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3269 -
accuracy: 0.8673
Epoch 85/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3272 -
accuracy: 0.8651
```

```
Epoch 86/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3277 -
accuracy: 0.8639
Epoch 87/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3269 -
accuracy: 0.8670
Epoch 88/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3268 -
accuracy: 0.8669
Epoch 89/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3277 -
accuracy: 0.8660
Epoch 90/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3272 -
accuracy: 0.8665
Epoch 91/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3269 -
accuracy: 0.8661
Epoch 92/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3278 -
accuracy: 0.8658
Epoch 93/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3280 -
accuracy: 0.8654
Epoch 94/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3274 -
accuracy: 0.8650
Epoch 95/100
667/667 [==============================] - 2s 2ms/step - loss: 0.3278 -
accuracy: 0.8648
Epoch 96/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3270 -
accuracy: 0.8633
Epoch 97/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3275 -
accuracy: 0.8648
Epoch 98/100
667/667 [==============================] - 2s 3ms/step - loss: 0.3272 -
accuracy: 0.8659
Epoch 99/100
667/667 [==============================] - 3s 4ms/step - loss: 0.3272 -
accuracy: 0.8665
Epoch 100/100
667/667 [==============================] - 3s 4ms/step - loss: 0.3274 -
accuracy: 0.8659
```

```
[34]: <tensorflow.python.keras.callbacks.History at 0x11dad1c2bb0>
```

Epoch 100/100 667/667 [==============================] - 3s 4ms/step - loss: 0.3274 - accuracy: 0.8659

1.4 Making the predictions and evaluating the model

1.4.1 Predicting the result of a single observation

Therefore, our ANN model predicts that this customer stays in the bank!

Important note 1: Notice that the values of the features were all input in a double pair of square brackets. That's because the "predict" method always expects a 2D array as the format of its inputs. And putting our values into a double pair of square brackets makes the input exactly a 2D array.

Important note 2: Notice also that the "France" country was not input as a string in the last column but as "1, 0, 0" in the first three columns. That's because of course the predict method expects the one-hot-encoded values of the state, and as we see in the first row of the matrix of features X, "France" was encoded as "1, 0, 0". And be careful to include these values in the first three columns, because the dummy variables are always created in the first columns.

Use our ANN model to predict if the customer with the following informations will leave the bank:

- Geography: France
- Credit Score: 600
- Gender: Male
- Age: 40 years old
- Tenure: 3 years
- Balance: $ 60000,
- Number of Products: 2,
- Does this customer have a credit card ? Yes
- Is this customer an Active Member: Yes
- Estimated Salary: $ 50000

So, should we say goodbye to that customer ?

- Assumption: Threshold Probability = 0.5
- Less than 0.5 = False. The person will not leave the bank
- More than 0.5 = True. The person will leave the bank

[39]:
```
print(ann.predict(sc.transform([[1, 0, 0, 600, 1, 40, 3, 60000, 2, 1, 1,␣
↪50000]])) > 0.5)
```

```
[[False]]
```

[40]:
```
print(ann.predict(sc.transform([[1, 0, 0, 600, 1, 40, 3, 60000, 2, 1, 1,␣
↪50000]])))
```

```
[[0.07359219]]
```

- Geography: Spain —> [0,1,0]
- Credit Score: 750
- Gender: Female
- Age: 35 years old

- Tenure: 10 years
- Balance: $ 70000,
- Number of Products: 4,
- Does this customer have a credit card ? Yes
- Is this customer an Active Member: No
- Estimated Salary: $ 50000

[36]:
```python
print(ann.predict(sc.transform([[0, 1, 0, 750, 1, 35, 10, 70000, 4, 1, 0,␣
↪50000]])) > 0.5)
```

```
[[ True]]
```

[41]:
```python
print(ann.predict(sc.transform([[0, 1, 0, 750, 1, 35, 10, 70000, 4, 1, 0,␣
↪50000]])))
```

```
[[0.8739985]]
```

1.4.2 Predicting the Test set results

[19]:
```python
y_pred = ann.predict(x_test)
y_pred = (y_pred > 0.5)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.
↪reshape(len(y_test),1)),1))
```

```
[[0 0]
 [0 1]
 [0 0]
 …
 [0 0]
 [0 0]
 [0 0]]
```

1.4.3 Making the Confusion Matrix

[20]:
```python
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test, y_pred)
```

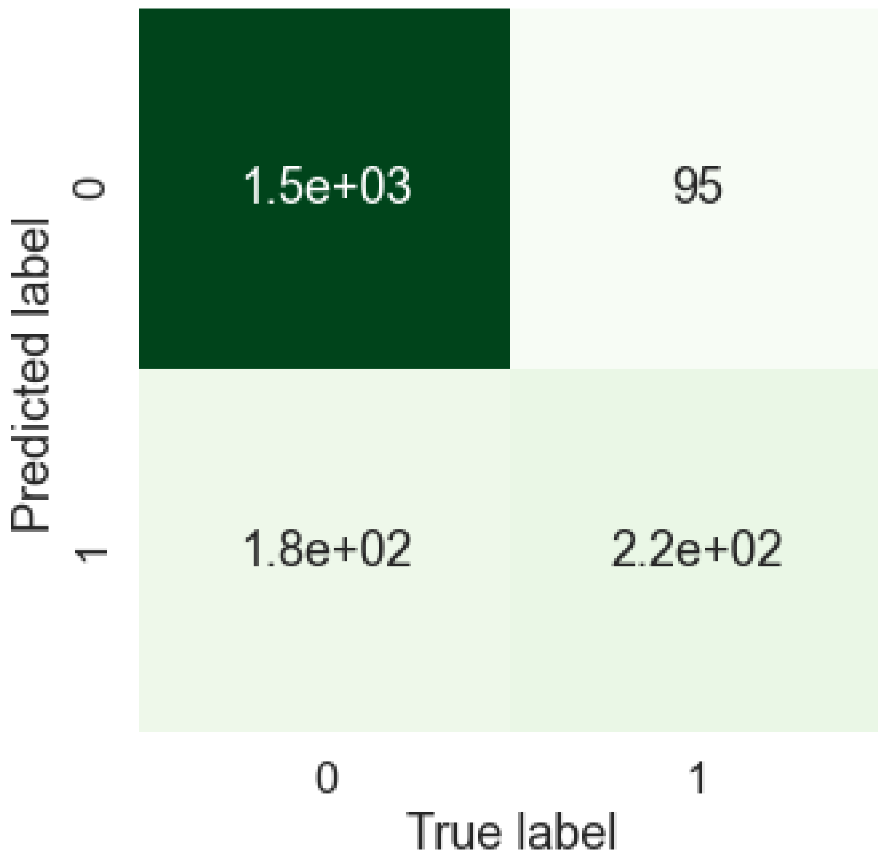
```
[[1500   95]
 [ 183  222]]
```

[20]: 0.861

[49]:
```python
import matplotlib.pyplot as plt
def plot_conf_mat(conf_mat):
    """
    Function that Plots a confusion matrix using seaborn's heatmap()
    """
    fig, ax = plt.subplots(figsize = (5,5))
```

```
    ax = sns.heatmap(conf_mat, annot=True, cbar = False, cmap="Greens")
    plt.xlabel("True label")
    plt.ylabel("Predicted label");

plot_conf_mat(cm)
```



[ ]:

[ ]:

[ ]:

[ ]:

[ ]: