Artificial Neural Network for TRS Pricing

I – Introduction

In this paper we analyse an alternative method using artificial neural network (ANN) to price Total return swap (TRS) looking at the efficiency of the ANN to learn the risk-neutral expectation TRS pricing formulas.

Author: Pierre Moureaux

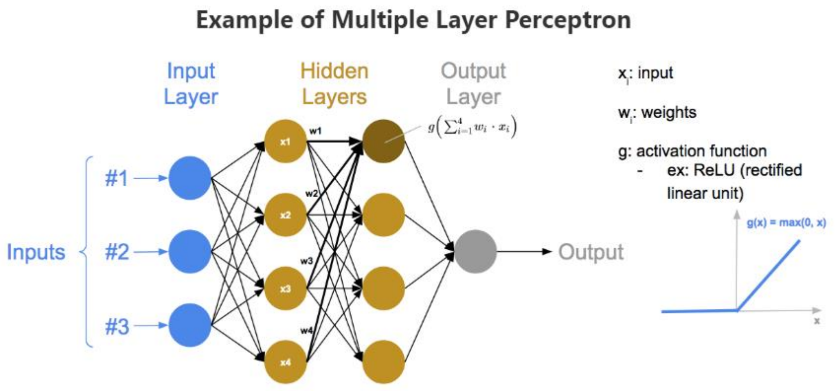
II – Artificial neural network

An artificial neural network (ANN) is a computational system inspired by the biological neural network found in animal brains. It consists of a collection of interconnected nodes, called neurons, which are organised into layers.

The neurons in each layer receive weighted inputs, process them through an activation function, and then pass the results to the neurons in the next layer. The neurons in the input layer receive input signals from the external environment while the neurons in the output layer produce the output. Hidden layers are intermediate layers between input and output layers.

The connections between neurons are weighted, which means that some inputs have a stronger influence on the output of the neuron than others. These weights are adjusted during training to improve the performance of the network for a specific objective.

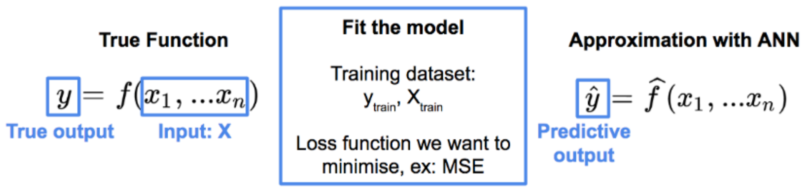
Below is an example of a multiple layer perceptron with three neurons in the input layer, two hidden layers, each of them having four neurons, and one neuron in the output layer.



The rectified linear activation function or ReLU is a piecewise linear function that will output the input if it is positive and zero otherwise. It is widely used as the default activation function in neural networks due to its ability to facilitate easier training and often yielding good performance.

ANN is an example of supervised learning algorithm; it can be used for both classification and regression.

Here we want to use an ANN to approximate a function.



We know the true function f, which will return the true output y from inputs X. We want to approximate it with the ANN.

From the input X, it will return a predictive output that we can compare to the true value.

To fit the model, we will define a training dataset where the model will learn, using a loss function to measure the difference between the estimation from the model and the true value, we want to minimize it. The mean squared error MSE is an example of loss function.

We use here the Adam algorithm (Adaptive Moment Estimation) to minimize the loss function. It is a popular optimization algorithm used in deep learning and particularly for training neural networks. It is an extension of the stochastic gradient descent optimization algorithm.

Two important inputs in the algorithm are the epoch and the batch size, they are both related to how the model learns from the data and they must be chosen carefully.

The Batch size is the number of training examples used in one iteration of the optimization algorithm. If the batch size is set to 16 as in our example, the model will take 16 data points at a time and update the weights of the model based on the average loss of those 16 samples. A larger batch size leads to a more stable training process but requires more memory to process. They are in general chosen to be a power of 2 as CPU and GPU memory architectures are organized in powers of 2, and it can be faster and more efficient to do so.

An epoch corresponds to the number of times the algorithm sees the entire training dataset. For example, if we have 750 data points in our training data set and a batch size of 16, then one epoch would involve 47 iterations (750 / 16 = 46.875). Training more epochs can improve the accuracy of the model but it can also lead to overfitting, so it is important to find a good balance.

III – TRS pricing with artificial neural network

The sample TRS will be, for the exercise, a very simple one, and the pricing framework will be simplified too:

- Bullet performance and financing leg
- Fixed financing rate
- The risk-free rate (the discounting rate) is assumed to be constant, which leads to simplified discount factors expression.

For all formulas, S represents the underlying price and M represents the bank-account value.

The market value of a TRS, using risk-neutral expectation, is calculated following fundamental theorem of pricing:

$$\frac{TRS(t_0,T)}{M(t_0)} = \mathbb{E}^{\mathbb{Q}}\left(\frac{(S_T - K) \; - \; r_{TRS} \cdot S_0 \cdot DCF(t_0,T)}{M(T)}\right)$$

$$\Rightarrow TRS(t_0,T) = \; S_0 - (K + \; r_{TRS} \cdot S_0 \cdot DCF(t_0,T))e^{-rT}$$

As $\frac{S}{M}$ is a martingale under $\mathbb{Q}$, and with additional following notations:

- K the strike, also called TRS settlement price
- r_{TRS} the TRS financing rate, equivalent, to some extent, to -repo rate
- $DCF(t_0,T)$ the day count fraction
- r the risk-free rate
- t_0 is assumed to be today, which leads to $M(t_0) = 1$

The TRS market value is a function of five variables, the asset price S, the strike price K, the time to maturity capital T minus t_0 , the TRS financing rate and the risk-free interest rate r.

As the TRS market value is linear homogenous in S and K, it can be reduced to three variables, fixing S at 100 for example.

We define the TRS market value of in the risk-neutral expectation framework, and the next python code highlights the TRS class (the reader will also find in github page the C++ code, using openNN library):

```python
#A very simple and vanilla bullet fixed-rate TRS class
class TRS:

    def __init__(self, S, K, T, r, r_TRS,Type):
        self.S = S
        self.K = K
        self.T = T
        self.r = r
        self.r_TRS = r_TRS
        self.Type = Type
        self.mv = self.mv()

    def mv(self):
        mv_TRS = self.S - math.exp(-self.r *self.T)*(self.K + self.r_TRS*self.S*self.T)
        if self.Type == "Receive performance":
            return mv_TRS
        if self.Type == "Pay performance":
            return -mv_TRS
```

And we create the dataset to train the model, with many values for the four variables and the corresponding TRS market values.

```python
data = []
for r_ in r:
    for Strike_ in Strike:
        for T_ in T:
            for r_TRS_ in r_TRS:
                data.append([r_, Strike_, T_, r_TRS_, \
                             TRS(100, Strike_, T_, r_, r_TRS_, "Receive performance").mv])
data = np.asarray(data)
```

Again, we split the dataset between training and test.

```
#training and test datasets
X = data[:,:4] #params r, strike, T, r_TRS
y = data[:,4:5] #TRS market value
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
```

We create and we fit the neural network on the training dataset. We use again a neural network with four layers, 10 neurons in each, the input dimension is now 4.

```
#ANN with four layers, 10 neurons each
#activation function: ReLU
ANN = Sequential()
ANN.add(Dense(10,input_dim = 4, activation = 'relu'))
ANN.add(Dense(10, activation = 'relu'))
ANN.add(Dense(10, activation = 'relu'))
ANN.add(Dense(10, activation = 'relu'))
ANN.add(Dense(1))
```

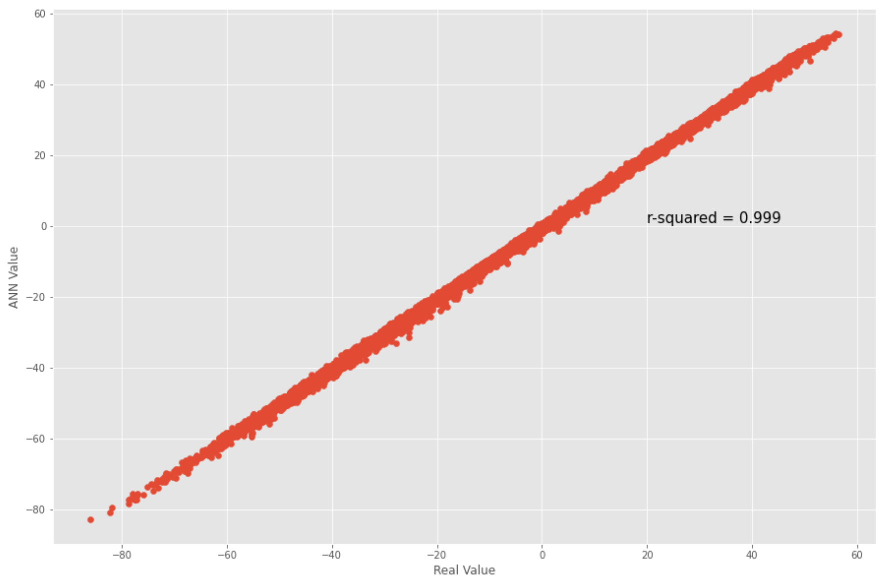
We did not change the number of epochs (150) or the batch size (16). Of course, the training takes more time compared to the previous simple example as the training dataset is bigger, the function being more complex with four variables instead of one.

```
#Loss function = MSE, optimizer: Adam
ANN.compile(loss = 'mean_squared_error', optimizer='adam')
# fit the ANN on the training dataset
ANN.fit(X_train, y_train, epochs = 150, batch_size = 16)
```

We still get good results when comparing estimated and real values in the test dataset.

```
#prediction
y_pred = ANN.predict(X_test)

#Comparison real values and predictions on test dataset
plt.figure(figsize = (15,10))
plt.scatter(y_test, y_pred)
plt.xlabel("Real Value")
plt.ylabel("ANN Value")
plt.annotate("r-squared = {:.3f}".format(r2_score(y_test, y_pred)), (20, 1), size = 15)
plt.show()
```

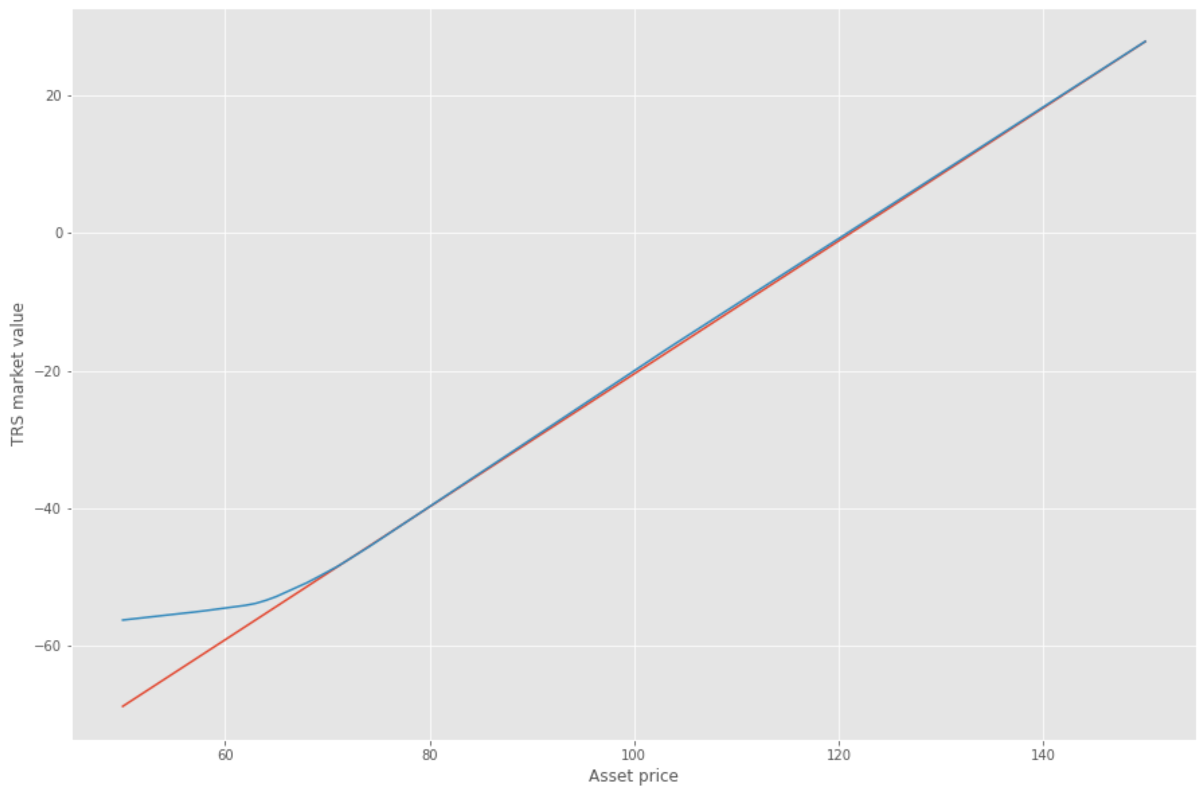


Now, we compare the market value of a TRS with a strike price at 120, a time to maturity at 6 months (0.5), the risk-free interest rate at 5% and the financing rate at 7% when changing the asset price.

```
K = 120 #strike price
r = 0.05 #risk-free interest rate
r_TRS = 0.07 #TRS financing rate
T = .5 #time to maturity
S = np.arange(50, 151, 1) #asset prices

PriceTheo = [TRS(S_, K, T, r, r_TRS, "Receive performance").mv for S_ in S]
PriceANN = [S_ / 100 * \
            ANN.predict(np.array([[r, K / S_ * 100, T, r_TRS]]))[0][0] for S_ in S]

#Comparison BS vs ANN prices
plt.figure(figsize = (15,10))
plt.plot(S, PriceTheo, label = "Theoretical market value")
plt.plot(S, PriceANN, label = "ANN market value")
plt.xlabel("Asset price")
plt.ylabel("TRS market value")
plt.show();
```



Moreover, we see on the left side of the chart that the model market value of the TRS is higher than straight red line (theoretical price) when the asset price decreases which is wrong for a delta one product like TRS (here receiver performance).

IV – Critics and open areas

- The highlighted TRS is a very vanilla one, the model should be extended to exotics TRS (underlying as hybrid basket, callability features, several payments etc…)
- The watchful reader may directly see that previous results are pure exercise and are in fact useless, as the TRS market value highlights a closed-form formula, which is of course far better to use than a complex ANN (and please note that this comment can be extent to all derivatives model which embeds closed-form formula, the utility of ANN in production is quite doubtful in those cases)
- The reader may also notice that we highlighted market value instead of price, which is, in TRS context, the fair financing rate which makes the market value equal to zero, but it's just a matter of additional method in TRS class, the model itself won't change.