Machine Learning - Practical Lesson 8

Matteo Sani
matteosan1@gmail.com

November 12, 2019

1 Basic Principles of Neural Networks

1.1 Overview

In this lesson we will see how machine learning techniques can be successfully applied to solve financial problems. We will first do a quick tour on the theory behind neural networks and then we will see an example and two practical applications regarding regression and classification issues.

1.2 Neural networks

1.2.1 Definition

Artificial Neural Networks (ANN) are information processing models that are developed by inspiring from the working principles of human brain. Their most essential property is the ability of learning from sample sets. The basic process units of ANN architecture are neurons which are internally in connection with other neurons.

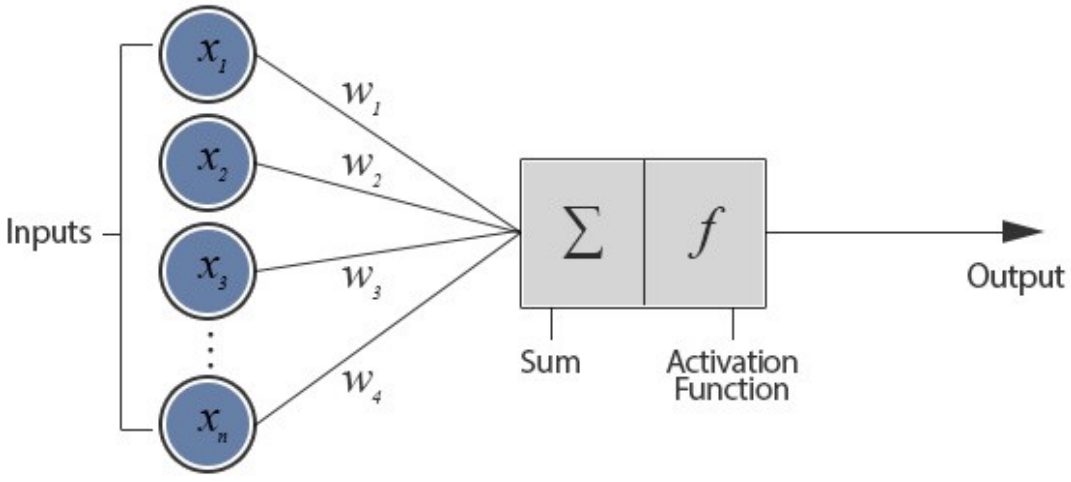


Figure 1: Model of an artificial neuron.

A neuron consists of weights (w_i) and real (x_i) numbers. All inputs injected into neurons are individually weighted, added together and passed into the activation function. There are many

different types of activation function but one of the simplest would be step function (another is the sigmoid).

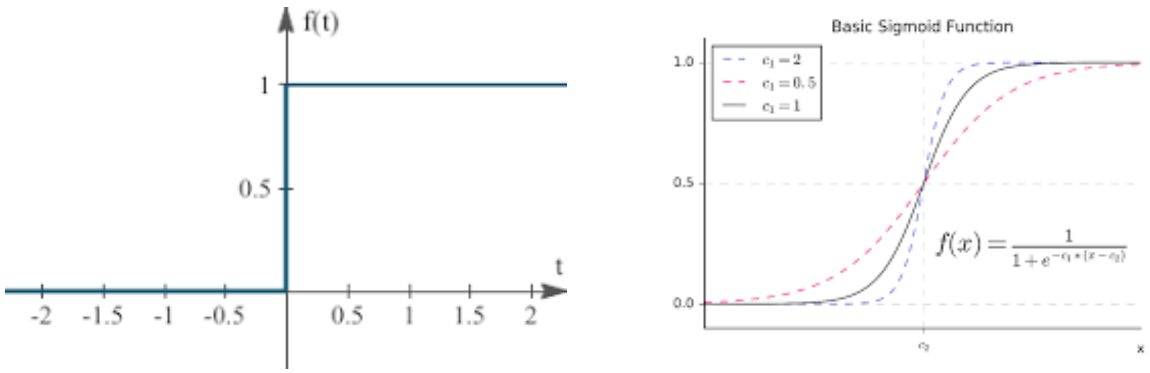


Figure 2: Step (left) and sigmoid (right) function.

The activation function is then responsible to provide the neuron output.

1.2.2 Training of a neuron

When teaching children how to recognize a bus, we just tell them, showing an example: "This is a bus. That is not a bus." until they learn the concept of what a bus is. Furthermore, if the child sees new objects that she hasn't seen before, we could expect her to recognize correctly whether the new object is a bus or not. This is exactly the idea behind the neurons. Similarly, inputs from a *training* set are presented to the neuron one after the other and weights are modified according to the expected output.

When an entire pass through all of the input training vectors is completed the neuron has learnt ! At this time, if an input vector $\vec{P}$ (already in the training set) is given to the neuron, it will output the correct value. If $\vec{P}$ is not in the training set, the network will respond with an output similar to other training vectors close to $\vec{P}$.

Unfortunately using just a neuron is not too useful since it is not possible to solve the interesting problems we would like to face with just that simple architecture. The next step is then to put together more neurons in *layers*.

1.2.3 Multi-layered neural networks

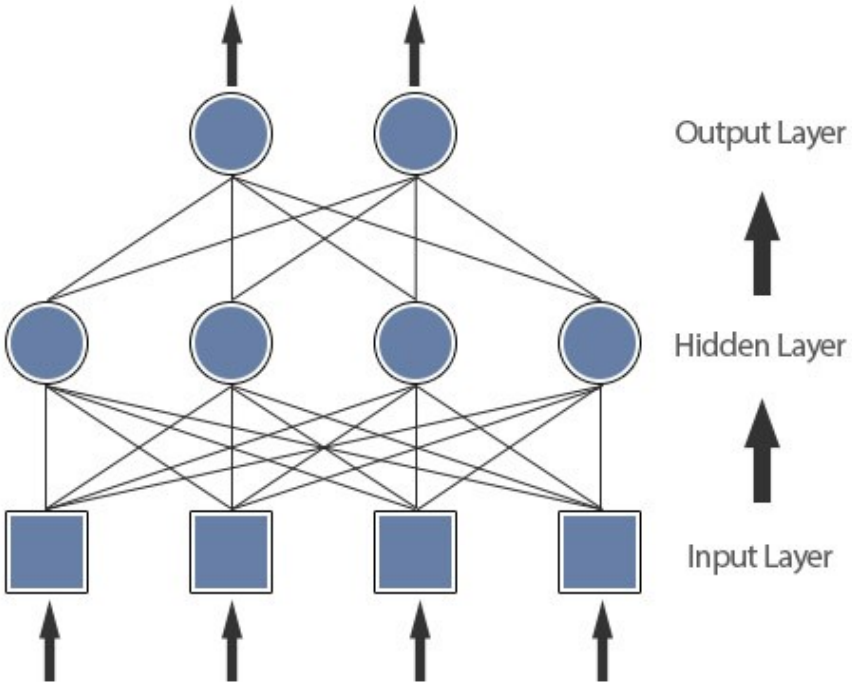


Figure 3: A multi-layered neural network.

Each input from the *input layer* is fed up to each node in the hidden layer, and from there to each node on the output layer. We should note that there can be any number of nodes per layer and there are usually multiple hidden layers to pass through before ultimately reaching the output layer. But to train this network we need a learning algorithm which should be able to tune not only the weights between the output layer and the hidden layer but also the weights between the hidden layer and the input layer.

1.2.4 Back propagation

First of all, we need to understand what do we lack. To tune the weights between the hidden layer and the input layer, we need to know the error at the hidden layer, but we know the error only at the output layer (we know the correct output from the training sample and we also know the output predicted by the network.) So, the method that was suggested was to take the errors at the output layer and proportionally propagate them backwards to the hidden layer.

So, what we are doing is:

- we present a training sample to the neural network (initialized with random weights);
- compute the output received by calculating activations of each layer and thus calculate the error;
- having calculated the error, we readjust the weights such that the error decreases;
- we continue the process for all training samples several times until the weights are not changing too much.

1.2.5 Neural Network Design

There is no rule to guide us into the design of a neural network in terms of number of layers and neuron per layer. The most common strategy is a trail and error one where you finally pick up the solution giving the best accuracy.

A common mistake to avoid is to start with a too complex (with many layers and neurons) network which usually leads to *overtraining*. Overtraining is what happens when the NN learns too well the training sample but its performance degrade substantially in an independent testing sample.

A NN with just one hidden layer with a number of neurons averaging the inputs and outputs is sufficient in most cases. In the following we will use more complex networks just for illustration, no attempt in optimizing the layout has been done.

1.3 Neural net to recognize handwritten digits

We don't usually appreciate how tough a problem our visual system solve (consider that it involves 5 visual cortices containing 140 million neurons each), but the difficulties of visual pattern recognition become apparent if you attempt to write a computer program to recognize digits like those below.

Figure 4: The so-called MNIST training sample

Simple intuition about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a morass of exceptions and caveats and special cases so that it seems hopeless.

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits and then develop a system which can learn from those training examples. By increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy. So while I've shown just 100 training digits above, we could certainly build a better handwriting recognizer by using thousands or even millions or billions of training examples (**remember that neural nets are not so capable of extrapolating results, hence it won't recongnize a digit written in some strange way not included in the training sample !!!**).

Let's try to implement an ANN that is capable of recognizing handwritten digits. To start we need to install three new modules, the easiest way of doing that is to run Anaconda on you computers (repl.it is too slow in this case):

- open an anaconda-shell and type the following: `pip install keras, mnist, tensorflow`

`pip` is a very useful tool that allows to install new modules to your python libraries. Alternatively using the Anaconda GUI you should be able to install the packages using the *Environment* tab.

Our program will be based on a Convolutional Neural Network (CNN, will see later other two types of NN) which is designed for image/pattern recognition. It works essentially by applying on top of an image a series of filters (matrices) that works as edge detectors and with them it classifies the images according to their features.

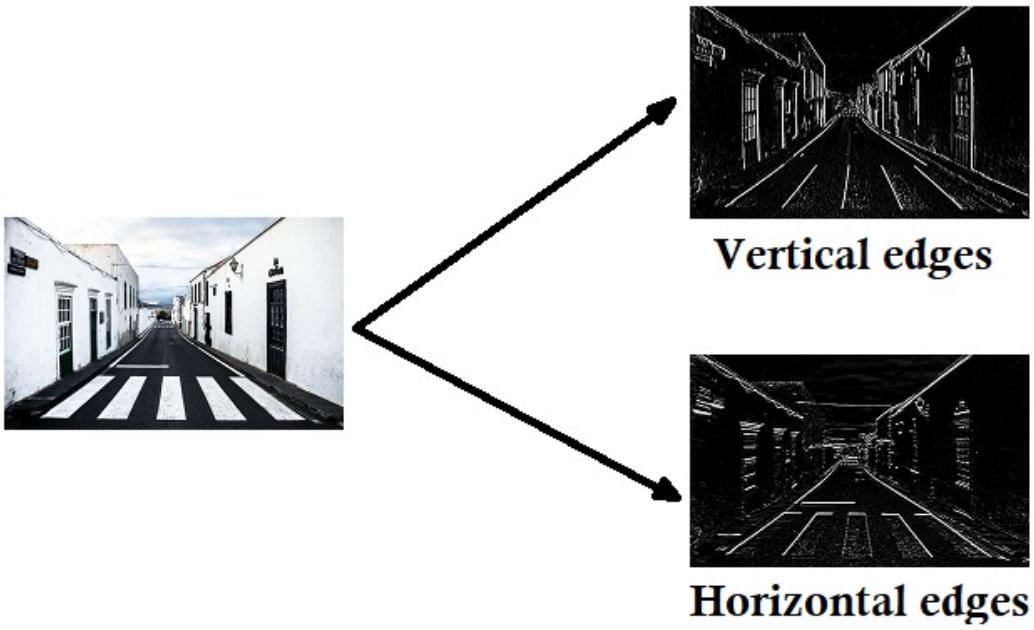


Figure 5: Filter application results on an image.

[5]:
```
import numpy as np
# contains our dataset for training
import mnist
# keras gives us all the tools to work with NN
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten
from keras.utils import to_categorical

# load the training
train_images = mnist.train_images() # the actual images
train_labels = mnist.train_labels() # the truth (it is a 0, 1, 2...)

# transform data for convenience
#train_images = (train_images / 255) - 0.5
# for technical reasons you need to expand axis
train_images = np.expand_dims(train_images, axis=3)

# definition of the actual network
```

```
num_filters = 8
filter_size = 3
pool_size = 2

# the input size reflects the size of the image with
# the numbers 28x28 pixels
model = Sequential([
    Conv2D(num_filters, filter_size, input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=pool_size), # reduce the size of the representation
                                       # to reduce the size of the parameters
    Flatten(), # this step of flattening is necessary to transform a
               # 2D matrix into a vector to connect to a classifier
    Dense(10, activation="softmax") # softmax is just another type of
                                    # activation functions like sigmoid or step␣
 ↪func.
])
# the output is given by 10 neurons returning the
# probability that image is in each class.

# adam is an algorithm to adjust the weights every cycle
# loss function compute the error between the prediction and the truth
# metrics which error to use
model.compile('adam', loss="categorical_crossentropy",
              metrics=['mean_squared_error'])

model.fit(train_images,
          to_categorical(train_labels),
          epochs=3)#,
          #validation_data=(test_images, to_categorical(test_labels)))

model.save('digit_training.h5')
```

```
Epoch 1/3
60000/60000 [==============================] - 9s 149us/step - loss: 1.8699 -
mean_squared_error: 0.0193
Epoch 2/3
60000/60000 [==============================] - 9s 155us/step - loss: 0.3434 -
mean_squared_error: 0.0092
Epoch 3/3
60000/60000 [==============================] - 9s 148us/step - loss: 0.2325 -
mean_squared_error: 0.0079
```

Let's try to see how well our NN predicts MNIST testing digits.

[5]:
```
import numpy as np
import mnist
from keras.models import load_model
```

```
model = load_model('digit_training.h5')

# testing with mnist test sample
test_images = mnist.test_images()
test_labels = mnist.test_labels()

test_images = np.expand_dims(test_images, axis=3)

predictions = model.predict(test_images[:5])
print ("Tesing on MNIST digits...")
print("Predicted: ", np.argmax(predictions, axis=1))
print("Truth:", test_labels[:5])
print("%:", ["{:.3f}".format(p[np.argmax(p)]) for p in predictions])
```

```
Tesing on MNIST digits...
Predicted:  [7 2 1 0 4]
Truth: [7 2 1 0 4]
%: ['1.000', '1.000', '0.995', '1.000', '1.000']
```

To see how well our NN behaves with different kind of digits we will try to check how it works with your own calligraphy.

- Open `paint` and create a 280x280 white square
- Change brush type and set the maximum size
- With the mouse draw a digit
- Finally save the file (e.g. five.png)

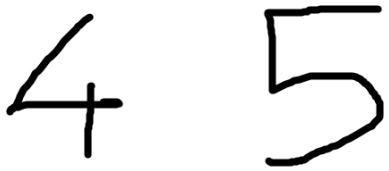
Before passing the image to the NN it has to be resized and this is done with an ad hoc function (`transform_image`).

[6]:
```
import numpy as np
from keras.models import load_model
from digit_converter import transform_image

model = load_model('digit_training.h5')

test_images = np.array(transform_image("five.png"))
test_images = np.expand_dims(test_images, axis=3)

predict = model.predict(test_images)
print ("\n")
print ("Tesing on custom digits...")
print ("Predicted: ", np.argmax(predict, axis=1))
print("%:", ["{:.3f}".format(p[np.argmax(p)]) for p in predict])
```

```
Tesing on custom digits...
Predicted:  [5]
```

```
%: ['0.739']
```

Those the images I have checked:



1.4 Black-Scholes call options

The first financial application of a NN concerns the pricing of european call options. In this case I have generated myself a large number of call options with a strike (100) and a maturity (1 year), simulated the underlying development and finally trained the NN using as inputs: volatility, strike, maturity and the underlying. The truth is the price of the call computed using the Black-Scholes formula.

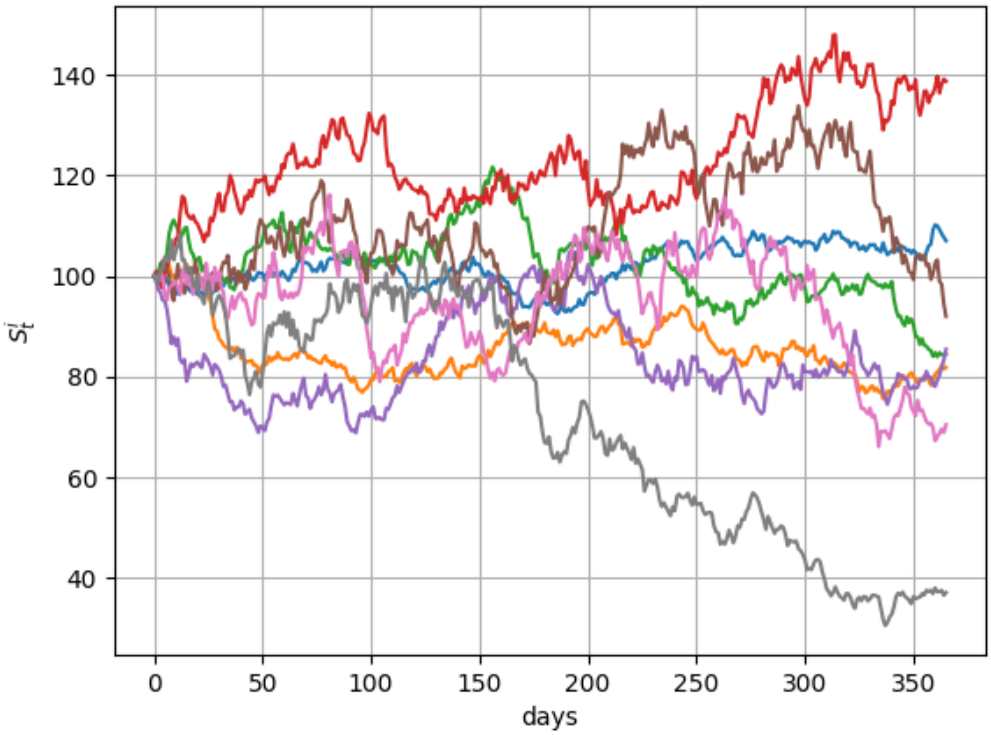


Figure 6: Some of the underlying development used in the BS simulation.

The code used for the simulation is in bs_simulation.py. I have also simulated two testing samples, one with the parameters included in the training events, and one with parameters outside. The

training and testing samples have been stored in a *csv* (comma-separated values) file, which has a particular format very easy to read.

In the training I have used a *traditional* NN with an input layer with 4 neurons (the number of inputs), an hidden layer with 10 neurons and an output layer with 1 single neuron (since I need just a number, the price of the call).

[7]:
```
# Regression Example
from keras.models import Sequential, load_model
from keras.layers import Dense
import pandas as pd

dataset = pd.read_csv("bs_training.csv")
X_train = dataset.iloc[:, :4].values
Y_train = dataset.iloc[:, 4].values

# create model
model = Sequential()
model.add(Dense(15, input_dim=4, kernel_initializer='normal', activation='relu'))
model.add(Dense(10, kernel_initializer='normal', activation='relu'))
model.add(Dense(5, kernel_initializer='normal', activation='relu'))
model.add(Dense(1, kernel_initializer='normal'))

model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['mse',␣
 ↪'mae'])

history = model.fit(X_train, Y_train, epochs=1000, verbose=1)#, batch_size=10)
evaluator = model.evaluate(X_train, Y_train)
print('Test: {}'.format(evaluator))

model.save('bs_model.h5')
```

```
Epoch 1/1000
2919/2919 [==============================] - 0s 90us/step - loss: 12.5533 - mse:
258.4865 - mae: 12.5533
...
Epoch 50/1000
2919/2919 [==============================] - 0s 49us/step - loss: 2.4020 - mse:
8.7222 - mae: 2.4020
...
Epoch 1000/1000
2919/2919 [==============================] - 0s 35us/step - loss: 0.1038 - mse:
0.0183 - mae: 0.1038
2919/2919 [==============================] - 0s 33us/step
Test: [0.08122403969150388, 0.011596422642469406, 0.08122401684522629]
```

Let's see now how the NN behaves with the two testing samples. First the one generated with parameters in the training phase space.

[9]:
```
from keras.models import load_model
import pandas as pd
import matplotlib.pyplot as plt

model = load_model('bs_model.h5')

dataset = pd.read_csv("bs_testing.csv")
X_test = dataset.iloc[:, :4].values
Y_test = dataset.iloc[:, 4].values

plt.plot(model.predict(X_test), color="red", label="NN price")
plt.plot(Y_test, label="BS price")
plt.legend()
plt.show()
plt.savefig("comparison_fair.png")
```

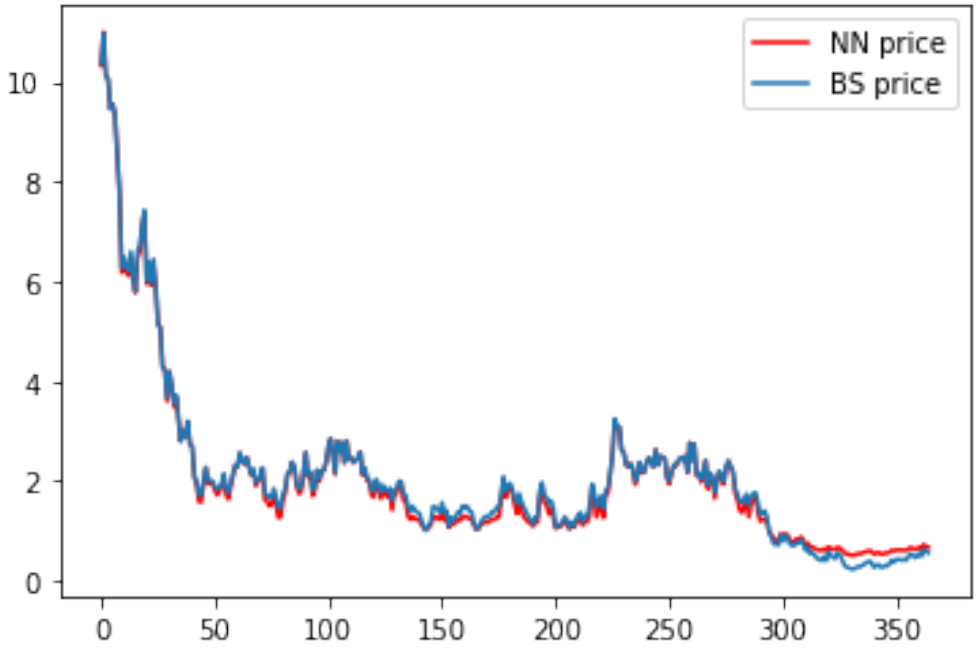


Figure 7: Comparison of neural network prediction (red) to BS pricing (blue).

The agreement is pretty good. To illustrate how a neural network is not able to extrapolate results if the prediction is tried with inputs outside the phase-space of the training (i.e. testing sample different from the one used in the training) I have tried to predict the price of a call with different maturity (strike and vol are in the range of the training instead):

[10]:
```
from keras.models import load_model
import pandas as pd
import matplotlib.pyplot as plt
```

```
model = load_model('bs_model.h5')

dataset = pd.read_csv("bs_testing_off.csv")
X_test = dataset.iloc[:, :4].values
Y_test = dataset.iloc[:, 4].values

plt.plot(model.predict(X_test), color="red", label="NN price")
plt.plot(Y_test, label="BS price")
plt.legend()
plt.show()
plt.savefig("comparison_off.png")
```

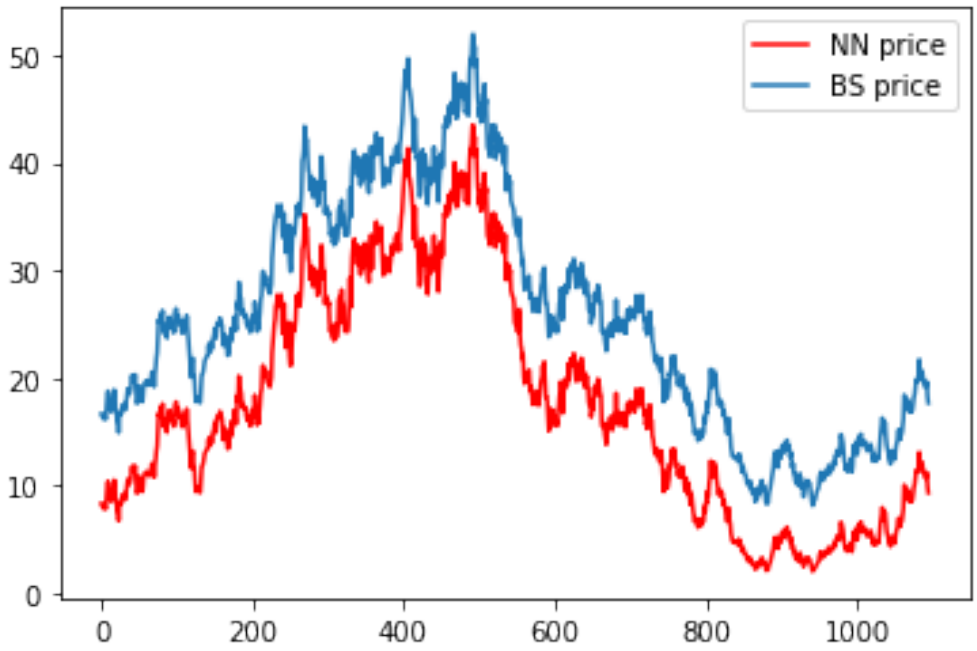


Figure 8: Comparison of neural network prediction (red) to BS pricing (blue) on a simulation done with parameter values outside the training phase-space.

1.5 Technical Analysis

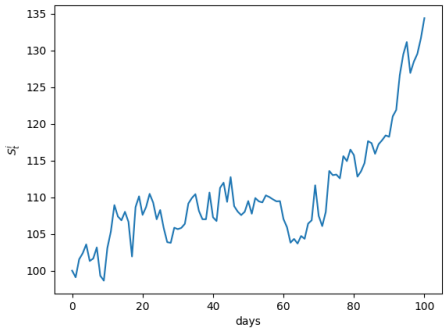
In finance, technical analysis is a security analysis discipline for forecasting the direction of prices through the study of past market data, primarily price and volume. Essentially the analyst looks for particular patterns in the price time series that are *known* to develop in predictable ways to take profit of it.

(a) Head and shoulders pattern in real data.

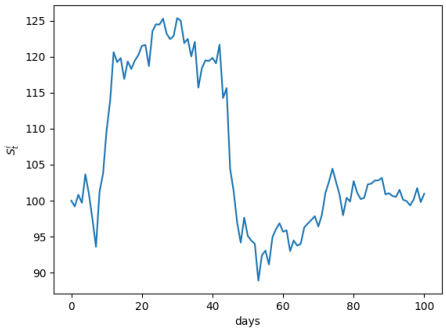
(b) Ascending triangle pattern in real data.

As you may imagine we will try to develop a CNN (like in the handwriting case) capable of classifying features in time series to be used in a technical analysis (this is much faster than having somebody looking at thousands of time series by eye...).

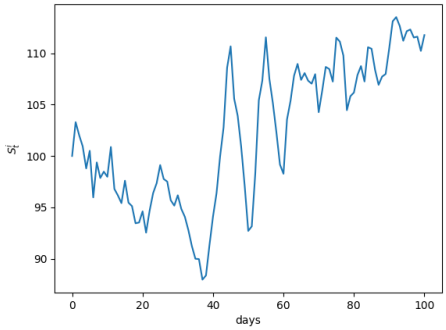
As in the previous application I have generated myself the training set simulating 21600 time series (1/3 with head and shoulder patter, 1/3 with triangle pattern and 1/3 with no pattern). *To make the training easier the features have been exagerated.*



(a) No pattern



(b) Head and shoulder pattern



(c) Tringle pattern

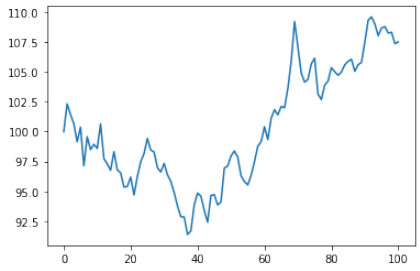
[20]:
```
import numpy as np
import json
from keras.models import Sequential, load_model
from keras.layers import Dense, Conv1D, Dropout, MaxPooling1D, Flatten,␣
 ↪GlobalAveragePooling1D
from keras.utils import to_categorical

# load the training set
with open("training_techana_labels.json", "r") as f:
    train_labels = json.load(f)
train_labels = train_labels[:3000]
train_images = []

with open("training_techana_images.json", "r") as f:
    train_images = json.load(f)
train_images = train_images[:3000]
train_images = np.array(train_images)
train_images = np.expand_dims(train_images, axis=3)

# define the CNN
model = Sequential()
model.add(Conv1D(filters=80, kernel_size=20,
                 activation='relu', input_shape=(101, 1)))
model.add(Conv1D(filters=80, kernel_size=15,
                 activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(filters=100, kernel_size=10,
                 activation='relu'))
model.add(Conv1D(filters=100, kernel_size=5,
                 activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
#model.add(Flatten())
model.add(Dense(3, activation="softmax"))
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])

# make the training
model.fit(train_images, to_categorical(train_labels),
          epochs=80, batch_size=35, verbose=2)

model.save('techana.h5')
```

```
Epoch 1/80
 - 2s - loss: 0.9404 - accuracy: 0.7007
...
 Epoch 40/80
```

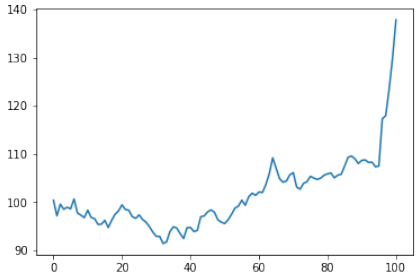
```
 - 2s - loss: 0.3310 - accuracy: 0.8697
...
 Epoch 80/80
 - 1s - loss: 0.1953 - accuracy: 0.9237
```

To test the perfomance I have created a longer time series and passed as input to the CNN a sliding time window to simulate the evolution of the price and a feature that is coming. The goal is to check when the neural net is capable of predicting the incoming pattern.

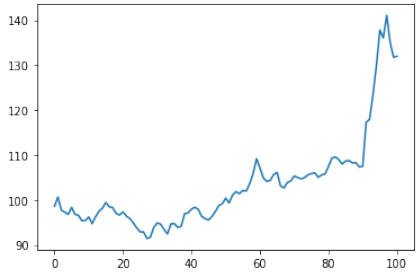
[2]:
```
import numpy as np
import json
from keras.models import Sequential, load_model
from keras.utils import to_categorical
from matplotlib import pyplot as plt

test_images = []

with open("testing_techana_frames.json", "r") as f:
    test_images = json.load(f)

test_images = np.array(test_images)
for i in range(test_images.shape[0]):
    plt.plot(test_images[i, :])
    plt.show()
test_images = np.expand_dims(test_images, axis=3)

model = load_model('techana.h5')

predictions = model.predict(test_images)
for i in range(len(predictions)):
    print (np.argmax(predictions[i]), max(predictions[i]))
```



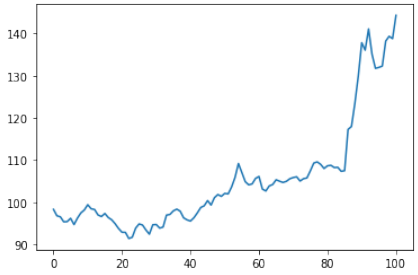
(a) Price at $T = 0$



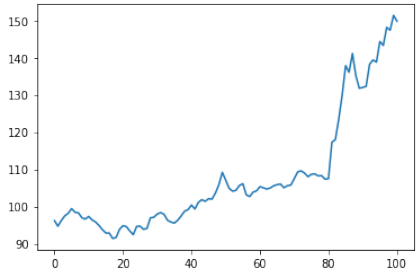
(b) Price at $T = 5$



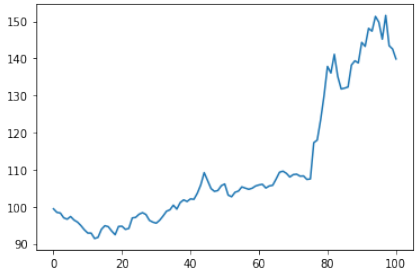
(c) Price at $T = 10$



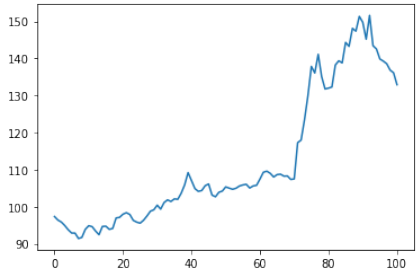
(d) Price at $T = 15$



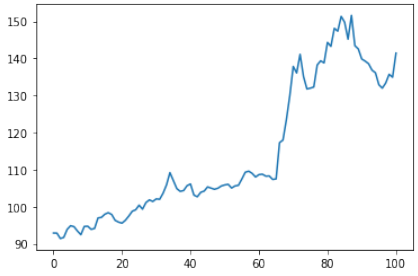
(e) Price at $T = 20$



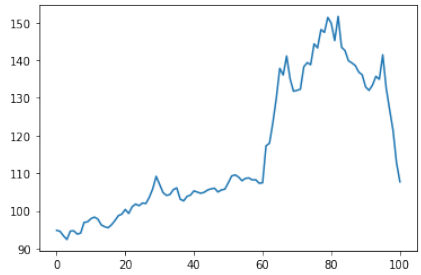
(f) Price at $T = 25$



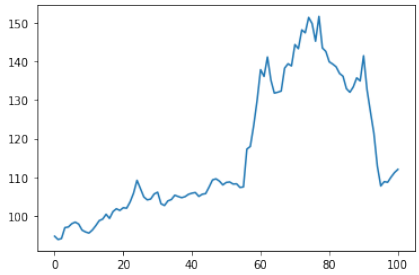
(g) Price at $T = 30$



(h) Price at $T = 35$



(i) Price at $T = 40$



(j) Price at $T = 45$

```
0  [0.96 0.00 0.03]
0  [0.66 0.00 0.33]
0  [0.96 0.00 0.03]
0  [0.93 0.00 0.06]
0  [0.93 0.00 0.06]
0  [0.56 0.00 0.43]
1  [0.00 1.00 0.00]
1  [0.00 1.00 0.00]
1  [0.00 1.00 0.00]
1  [0.00 1.00 0.00]
```

So at the 6th sample the CNN start recognizing the *head and shoulder* pattern in the price evolution.

1.6 Exercises

1.6.1 Exercise 8.1

Try to repeat all the example shown during the lesson.

1.6.2 Exercise 8.2

Using the same code illustrated above, test the ANN to recognize digits with your own handwriting (e.g. try to exagerate some feature to fool the NN, or even pass it letters instead of digits and interpret the results).

1.6.3 Exercise 8.3

Taking as example the pricing NN trained on call, try to price put options.