Q-Learning

Marin & Salim: **Trading Application**

Created: July, 2017

In this tutorial, we will present to you how to run the Deep Q Trading technique on a specific stock.

Requirements: Numpy Pandas MatplotLib scikit-learn TA-Lib, instructions at https://mrjbq7.github.io/ta-lib/install.html Keras, https://keras.io/ Quandl, https://www.quandl.com/tools/python

In [3]:

```
import pandas as pd
import numpy as np
import q_aux_functions #Auxiliaires functions to reduce the size of the cod
e
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop
import pandas_datareader.data as web
from sklearn import preprocessing
from pandas import datetime
from keras.optimizers import Adam
import quandl
from talib import abstract
from sklearn.externals import joblib
import backtest as twp #This is for the return and other things
import random, timeit
```

```
Using TensorFlow backend.
```

You need to install the previous requirements in order to run the code, moreover you will need the Tensorflow back-up to install Keras and the neural-network. You can find how to install Tensorflow at this page:

https://www.tensorflow.org/install/

This function will download the stock from the website QuanDL (you need to register), then it will split the stock into two parts: the train set (6 years) and the test set (last year).

In [4]:

```
stock_name = 'FAST'

quandl.ApiConfig.api_key = 'RUE-xjsauuMHrcGkVwSj' #Set up the QuanDL config
uration

def get_stock_data_quandl(stock_name):
    """
    Return a dataframe of that stock
```

```
    Return a dataframe of that stock.
    """
    df = quandl.get_table('WIKI/PRICES', ticker = stock_name) #Download the
prices from QuanDL
    df.drop(['ticker', 'adj_open', 'adj_high', 'adj_low', 'ex-dividend', 's
plit_ratio'], 1, inplace=True)
    df.set_index('date', inplace=True)

    # Renaming all the columns so that we can use the old version code
    #df.rename(columns={ 'adj_open': 'Open', 'adj_high': 'High', 'adj_low':
'Low', 'adj_volume': 'Volume',
    #'adj_close': 'Adj Close'}, inplace=True)

    X_train = df.iloc[-2000:-300,:] #Take the last 1700 values
    X_test = df.iloc[-300:, :] #Take the last 300 values for the train set
    return [X_train, X_test]
```

Then we initialize the data and create the first state S: [close, diff, sma15, close-sma15, sma15-sma60, rsi, atr].

In [5]:

```
#Initialize first state, all items are placed deterministically
def init_state(indata, test=False):
    close = indata['close'].values
    diff = np.diff(close) #Take the diff of the close values
    diff = np.insert(diff, 0, 0) #Insert 0 in the first position in order t
o
    sma15 = abstract.SMA(indata, timeperiod=15) #Take the simple mean
average for 15 days
    sma60 = abstract.SMA(indata, timeperiod=60)
    rsi = abstract.RSI(indata, timeperiod=14) #measures the speed and
change of price movements
    atr = abstract.ATR(indata, timeperiod=14) #indicator for the
volatibility of the market

    #--- Preprocess data
    xdata = np.column_stack((close, diff, sma15, close-sma15, sma15-sma60,
rsi, atr)) #Change 1D arrays to 2D array

    xdata = np.nan_to_num(xdata) #Delete the NaN values
    if test == False:
        scaler = preprocessing.StandardScaler() #Standard scaler (unit
variance and remove mean)
        xdata = np.expand_dims(scaler.fit_transform(xdata), axis=1)


        joblib.dump(scaler, 'data/scaler.pkl')
    elif test == True:
        scaler = joblib.load('data/scaler.pkl')
        xdata = np.expand_dims(scaler.fit_transform(xdata), axis=1)
    state = xdata[0:1, 0:1, :] #Take only the 7 first values of the first
day (i.e the first state)

    return state, xdata, close
```

We set up the different parameters and we also initialize the neural network:

In [6]:

In [6]:

```
tsteps = 1
batch_size = 1
num_features = 7 #Features = [close, diff, sma15, close-sma15, sma15-sma60,
rsi, atr] only 1 time-step

adam = Adam() #This is for the optimizer.
nb_actions = 3
neurons = [64, 32, 32, nb_actions] #Build the Model here
dropout = 0.3 #parameter, good values are 0.3 or 0.7
model = q_aux_functions.build_model(neurons, num_features, dropout)
model.compile(loss='mse', optimizer=adam)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 1, 64)             18432
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 64)             0
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                12416
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 99
=================================================================
Total params: 32,003
Trainable params: 32,003
Non-trainable params: 0
_________________________________________________________________
Inputs: (None, 1, 7)
Outputs: (None, 3)
```

Then we run the main loop with a number of epochs given (in this example the number is 3 and it takes 15 minutes approximately). Each epoch, train our neural netowrk everyday in order to approximate the Q-function, we take an action everyday and then calculate the reward associated:

In [7]:

```
#Take Action
def take_action(state, xdata, action, signal, time_step):
    #this should generate a list of trade signals that at evaluation time a
re fed to the backtester
    #the backtester should get a list of trade signals and a list of price
data for the assett

    #make necessary adjustments to state and then return it
    time_step += 1

    #if the current iteration is the last state ("terminal state") then set
terminal_state to 1
    if time_step + 1 == xdata.shape[0]:
        state = xdata[time_step-1:time_step, 0:1, :]
        terminal_state = 1 #It's over now
        signal.loc[time_step] = 0

        return state, time_step, signal, terminal_state
```

```
    #move the market data window one step forward
    state = xdata[time_step-1:time_step, 0:1, :]
    #take action
    if action == 1:
        signal.loc[time_step] = 100 #Buy
    elif action == 2:
        signal.loc[time_step] = -100 #Sell
    else:
        signal.loc[time_step] = 0 #Hold
    #print(state)
    terminal_state = 0
    #print(signal)

    return state, time_step, signal, terminal_state
```

Then we can run the main loop. For each epoch, it will run it for every day and calculate the action and the associated reward, then it will update the neural network in order to try to fit to the Q function value.

In [ ]:

```
```

In [ ]:

```
while(status == 1):
        #We are in state S
        #Let's run our Q function on S to get Q values for all possible act
ions
        qval = model.predict(state, batch_size=1) #Returns a 3 array with
value for each action = [Buy, Sell, Hold]
        if i == epochs-1:
            evolution_of_q.append(qval) #Show the evolution of the Q-value
only for the last day
        if (random.random() < epsilon): #choose random action (explore)
            action = np.random.randint(0, 3) #assumes 3 different actions
        else: #choose best action from Q(s,a) values
            action = (np.argmax(qval))
        #Take action, observe new state S'
        new_state, time_step, signal, terminal_state = take_action(state, xd
ata, action, signal, time_step)
        #Observe reward (it's a value)
        reward = get_reward(new_state, time_step, action, price_data, signal
, terminal_state)

        #Experience replay storage
        if (len(replay) < buffer): #if buffer not filled, add to it
            replay.append((state, action, reward, new_state))
            #print(time_step, reward, terminal_state)
```

This is the main part of the algorithm. We start from the first state (i.e close values of the day 1). Then we use the epsilon-greedy strategy: we can decide to choose a random action (exploration) or to simply take the action which has the best Q-value (which means it's the best action to perform).

Imagine the action is 'buy', then we perform this action and then we are now in the second state (i.e the following day).

In [ ]:

```
def get_reward(new_state, time_step, action, xdata, signal, terminal_state,
eval=False, epoch=0):
    reward = 0
    signal.fillna(value=0, inplace=True)

    if eval == False:
        bt = twp.Backtest(pd.Series(data=[x for x in xdata[time_step-2:time_
step]], index=signal[time_step-2:time_step].index.values), signal[time_step
-2:time_step], signalType='shares')
        price_t_1 = bt.data['price'].iloc[-1] #Get the last price p(t-1)
        price_t_2 = bt.data['price'].iloc[-2]
        reward = bt.data['shares'].iloc[-2] * ((price_t_1 - price_t_2) / pri
ce_t_2 )
```

We use this function to obtain the reward, in our case, the reward is the money (earn or lost) after performing the action. So if it was a good choice to 'buy' the reward will be positive and then the action will be keep for a future similar state.

In [ ]:

```
for memory in minibatch: #Browse the list of the minibatch
    #Get max_Q(S',a)
    old_state, action, reward, new_state = memory
    old_qval = model.predict(old_state, batch_size=1) #3 values: [buy,
hold, sell]
    newQ = model.predict(new_state, batch_size=1) #Predict Q(s', a)
    maxQ = np.max(newQ)  #Take max{ Q(s', a) }
    y = np.zeros((1,nb_actions)) # matrix size = (1, 3)
    y[:] = old_qval[:] #Take the set of actions
    if terminal_state == 0: #non-terminal state
        update = (reward + (gamma * maxQ))
    else: #terminal state
        update = reward
    y[0][action] = update #The action which corresponds is updated
    #print(time_step, reward, terminal_state)
    X_train.append(old_state) #The inputs is the 7 features
    y_train.append(y.reshape(nb_actions,)) #The output of the neural networ
k is 3

X_train = np.squeeze(np.array(X_train), axis=(1)) #Remove single dimensiona
l entries from a single array
#Shape it for the fitting
y_train = np.array(y_train) #Same thing
model.fit(X_train, y_train, batch_size=batchSize, epochs=1, verbose=0)
#Create a new fitting model
```

This part is the key to understand the Deep Q-Learning. We simply update the Q-value to the optimal Q function which can be approximate with the formula: (reward + (gamma * maxQ)).

Then we train the model (we fit it) to mimic the Q function to find. We do it a great number of times. At the end, the neural network has been trained to perform the best action given any state.

In [ ]:

```
if i == epochs-1: #the last epoch, use test data set
        state, xdata, price_data = init_state(X_test, test=True)
```

When the neural network is trained, we then test it over the last 300 days of the stock.

At the end of the loop (can be long depends on the GPU, in my case it approximately 5 minutes for 1 epoch), we display all the important informations, first we show the trade on the training set:

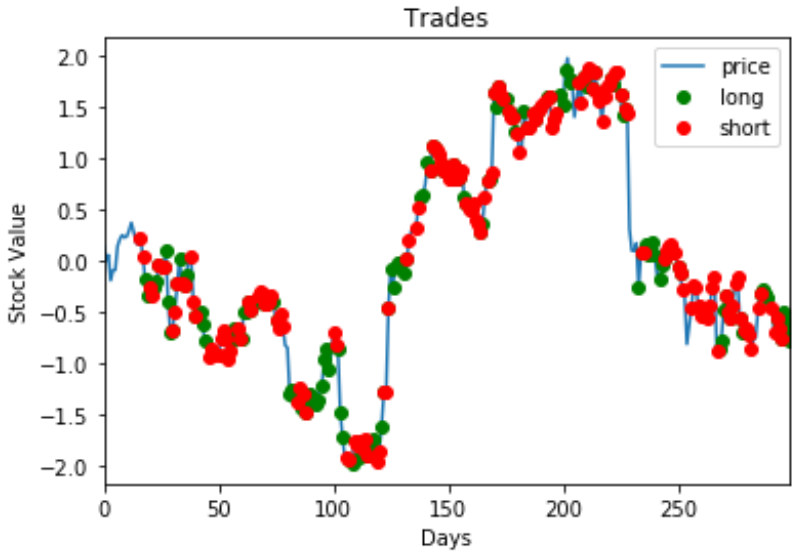
In [9]:

```
### GIVE INFORMATION ABOUT THE RETURN / PRICE ###

bt = twp.Backtest(pd.Series(data=[x[0,0] for x in xdata]), signal, signalTyp
e='shares')
bt.data['delta'] = bt.data['shares'].diff().fillna(0)

#print(bt.data)
unique, counts = np.unique(filter(lambda v: v==v, signal.values), return_cou
nts=True)
#print(np.asarray((unique, counts)).T)


bt.plotTrades() #Plot the different trades (Hold, Buy or Sold)
plt.show()
```

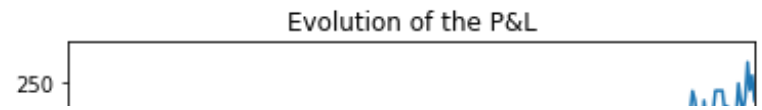


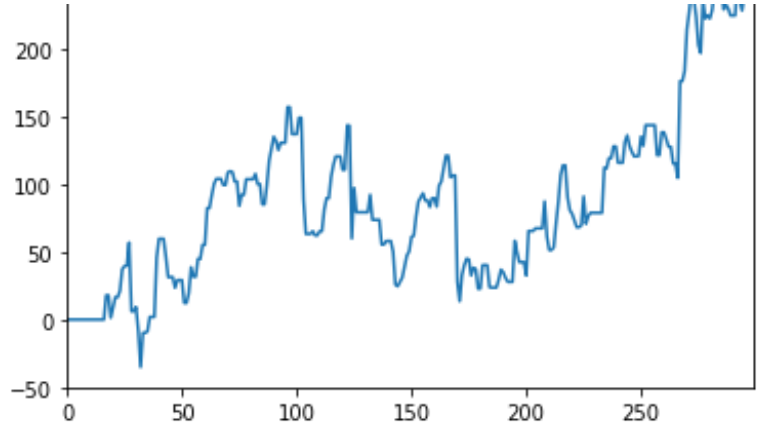
Then we plot the P&L, the stock and the cash that we have earned:

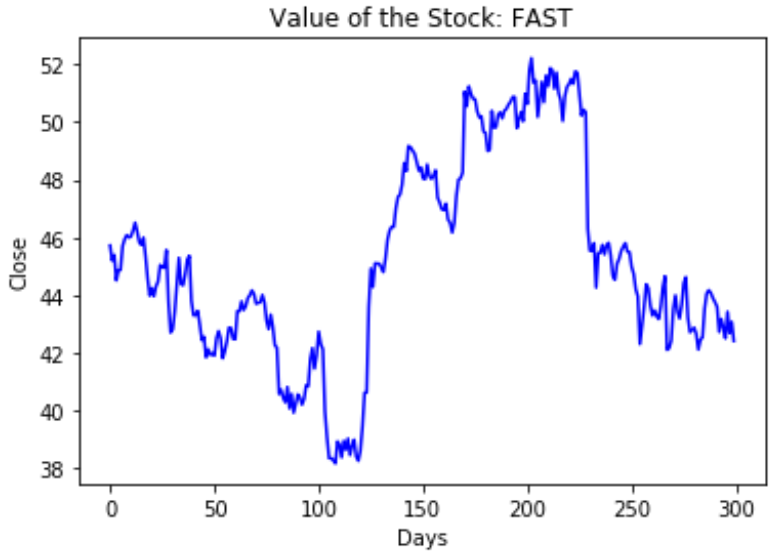
In [10]:

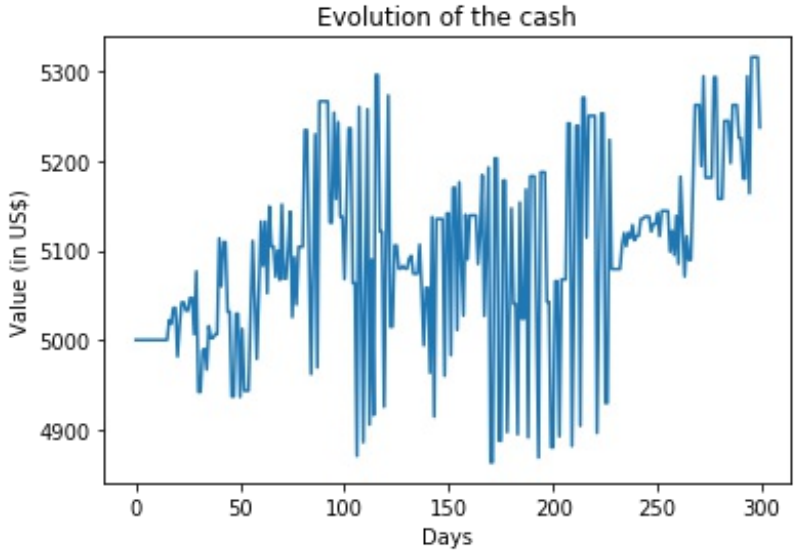
```
plt.title("Evolution of the P&L")
bt.pnl.plot() #Plot the PNL
plt.show()

q_aux_functions.plot_stock(stock_name, X_test) #Plot the stock for comparis
on
return_value = q_aux_functions.get_return(bt.data['cash']) #Get the return
of the cash and plot it.
```










```
The return value over 300 days is 4.74254395259 %.
```

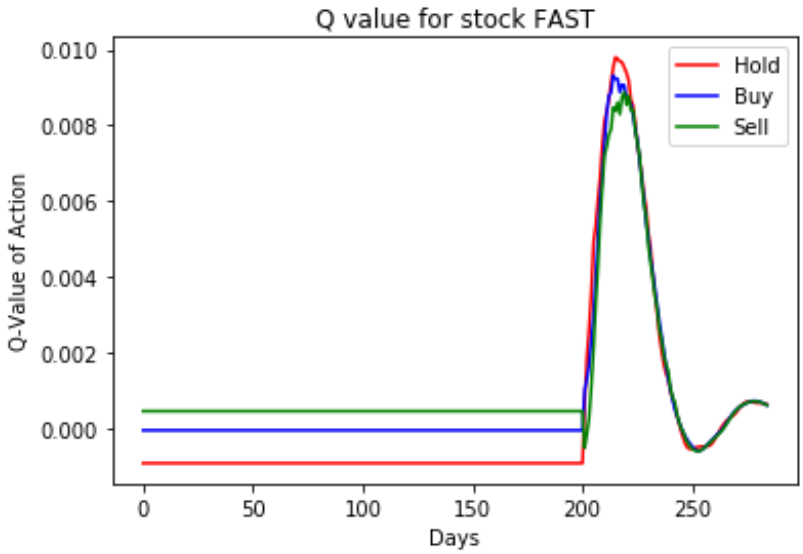
We also show some informations about the Q-learning, for example you can see the evolution of the Q-value for each day (and the action taken accordingly). If the number of epochs is superior than 8, we also plot the learning rate in order to see if the algorithm tends to converge or not.

In [12]:

```
#Plot the Q Value
q_aux_functions.plot_q_value(evolution_of_q, stock_name)

if 8 < epochs:
```

```
    plt.plot(learning_progress) #Show the evolution of the reward
    plt.show()
```

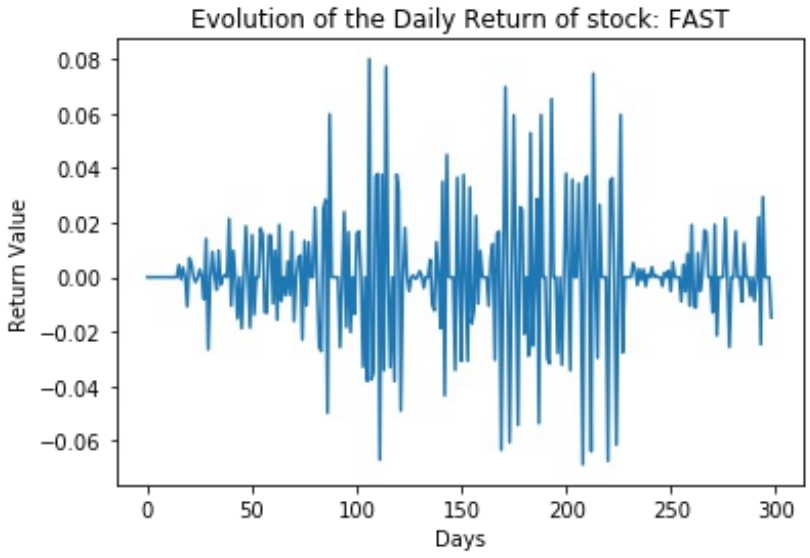


Finally we also show the return value for each day and we also compare it to a risk-free rate such as the T-bill in order to obtain the Sharpe Ratio:

In [13]:

```
q_aux_functions.calculate_daily_return(bt, stock_name) #Calculate and plot
the daily return

sharpe_ratio = q_aux_functions.calculate_sharpe_ratio(bt.data['cash'], 2016
) #Calculate the sharpe ratio given the year
```



We have also created an benchmark trading to compare to our trading strategy. This benchmark trading is called "monkey trading" and will perform a random action each day (buy / sell / hold). We run it 30 000 times and take the cash obtain for each, it shows us that our strategy is much better compare to this benchmark strategy.

In [16]:

```
def monte_carlo_monkey(nb_test, stock_name, start_cash, nb_of_days):
    """
        Run the Benchmark for many test, get the last value (or the return)
```

```
for each test.
        Then plot the histogram of the money earn.
    """
    histo = []
    for i in range(nb_test):
        value = benchmark_monkey(stock_name, start_cash, nb_of_days)
        #print(value)
        histo.append(value)

    histo = np.array(histo)
    mean_value = np.mean(histo)
    print("The mean value of the benchmark is", mean_value)
    plt.hist(histo)
    plt.title("Benchmark Monkey Histogram")
    plt.xlabel("Cash")
    plt.ylabel("Frequency")

plt.savefig('images/histogram_benchmark_'+str(nb_test)+'_simulation.png', b
box_inches='tight', pad_inches=1, dpi=72)
    return(histo)
```

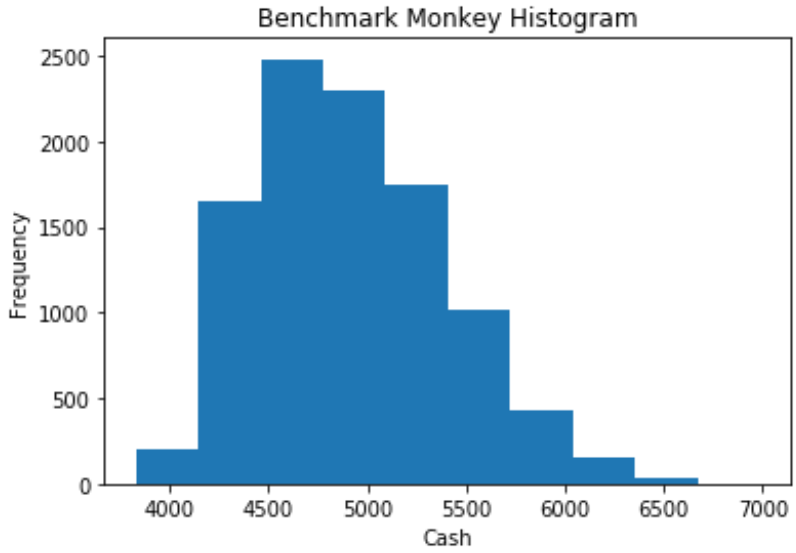
In [17]:

```
#Give the histogram of the trading (Benchmark)
#Test for 30 000 values (start_cash = 5 000$)
histo = monte_carlo_monkey(10000, "INTC", 5000, 300)
```

```
The mean value of the benchmark is 4918.8221
```

In [23]:

```
plt.hist(histo)
plt.title("Benchmark Monkey Histogram")
plt.xlabel("Cash")
plt.ylabel("Frequency")
plt.show()
```



Dueling Q Learning: A possible improvement of the Q-learning is to slighty change the neural network. Instead of having just one succeding layer, we can split it into two parts: the advantage and the value.

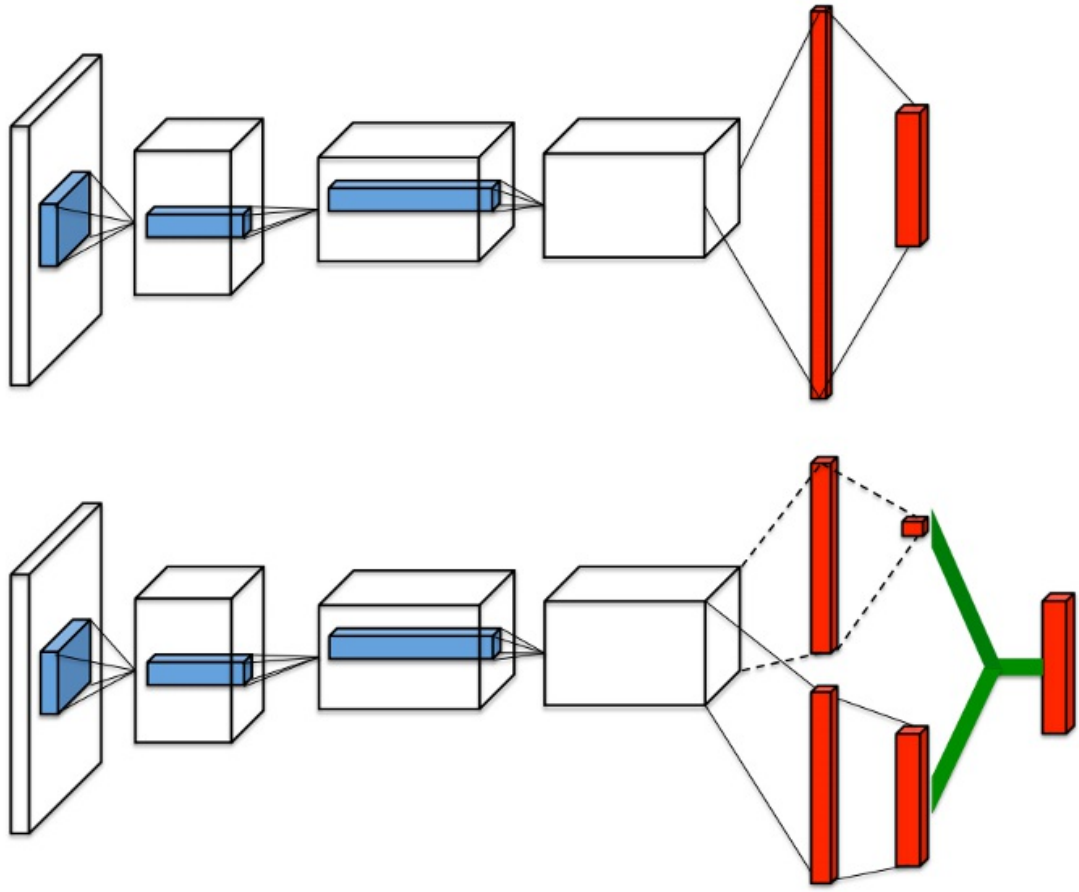
Actually, the value function tells us how the state that we are in is good, and the advantage layer will tell us which actions is the best to perform at that moment:

tell us which actions is the best to perform at that moment.

In [3]:

```
from IPython.display import Image
Image("pics/dueling_network.png")
```

Out[3]:



In [ ]:

```
def build_dueling_model(neurons, num_features, dropout, num_actions):
    """
        Build a Dueling Neural Network. We use Keras but in another way.
        It uses the architecture of the DeepMind paper.
    """
    model = Sequential()

    input_layer = Input(shape = (1, num_features))
    lstm1 = LSTM(neurons[0], input_shape=(1, num_features),
return_sequences=True)(input_layer)
    lstm2 = LSTM(neurons[1], input_shape=(1, num_features),
return_sequences=False)(lstm1)

    fc1 = Dense(neurons[2], kernel_initializer="uniform", activation='relu'
)(lstm2)
    advantage = Dense(num_actions)(fc1) #This is the layer dedicated to the
advantage function

    fc2 = Dense(neurons[2], kernel_initializer="uniform", activation='relu'
)(lstm2)
    value = Dense(1)(fc2) #This is the layer dedicated to the value
function
```

```
    policy = merge([advantage, value], mode = lambda x: x[0]-K.mean(x[0])+x[
1], output_shape = (num_actions,))
    #We merge it into the last layer with the formula dedicated

    model = Model(input=[input_layer], output=[policy])
    model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])
    model.summary()
    print("Inputs: {}".format(model.input_shape))
    print("Outputs: {}".format(model.output_shape))

    return(model)
```