Capstone Project

Machine Learning Engineer Nanodegree

Krishnan Ramaswamy

Reinforcement algorithms for trading systems

Table of Contents

1. Problem Statement

The definition of automated or algorithmic trading can be classified as following groups.

1. Market making. Mostly to control and drive the direction of market movement, which derived from the order books' information, to provide liquidity and profit from bid/ask spread from market makers' perspectives. Stochastic control in high frequency trading plays important role here.

2. Algorithmic trading. The motivation is to identify patterns and intelligent decision making like professional/experienced trader using combination of technical, fundamental and market sentiments.

Traders/investors uses many technical indicators and trading strategies which have been developed by researchers and hedge fund managers/companies. Also statistical and other computer aided machine learning approaches are prevalent in hedge fund industry. More recently, there are new cloud based services are being available – like https://stocksneural.net/ which helps in predicting the direction of stock

prices using machine learning/neural network techniques.

The objective of this project is to utilize machine learning/reinforcement techniques to identify market directors and corresponding decisions to BUY and SELL and provide better performance/risk than S&P

2. Proposed Solution

The objective of reinforcement learning based approach is not the minimization of the sum-of-squares error which is one of the objective of conventional supervised learning but the acquisition of an optimal policy under which the learning agent achieves the maximal average reward from the environment.

In this project, I am proposing an improvised reinforcement learning framework with many(flexible) indicators involved in prediction criteria with trading policies more effectively. The value approximator is trained using a regularizing technique for the prevention of divergence of the parameters with S&P. I am hoping to demonstrate a stock trading system implemented using the proposed framework will outperform the market average or individual stock being compared.

3. Data Sets and Inputs

There are various sources to get stock trading. We will use Yahoo finance to get

- S&P daily data for 15 years (involving both bear and bull markets)
- One can specify the ETF, stocks in instances.dict (explained in later sessions) and the framework dynamically get the data
- Daily data for about 10 ETFs and several large/med/small companies involving various industries

Note that the inputs will contain multiple metrics, such as opening price (Open), highest price the stock traded at (High), how many stocks were traded (Volume) and closing price adjusted for stock splits and dividends (Adjusted Close); your system only needs to predict the Adjusted Close price.

4. Benchmark

S&P index fund performance used as benchmark to compare the performance of various learners and strategies. In this project, we have implemented three learning agents and 3 types of strategies that utilize different combinations of learning agents.

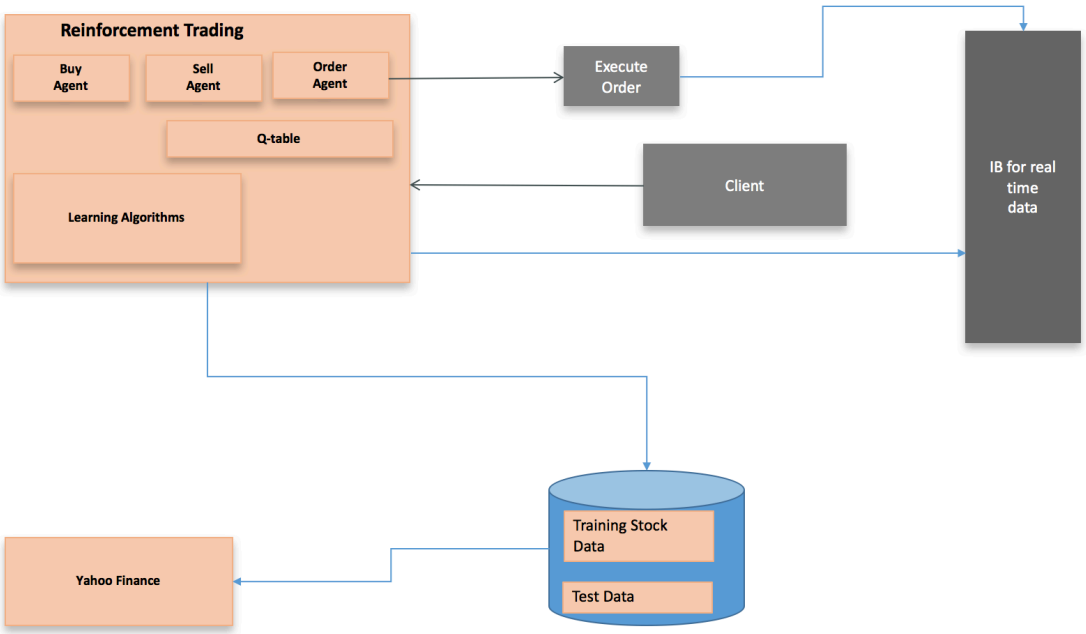
One can specify the benchmark data set as part of instance.dict. For example, following is an example in which SPY is specified as benchmark.

```
"instance2" : {
    "experiment"    : "experiment1",
    "tickerWeights" : {
                        "SPY": 0.20,
                        "IJS": 0.05,
                        "EFA": 0.15,
                        "EEM": 0.05,
                        "AGG": 0.20,
                        "JNK": 0.05,
                        "DJP": 0.10,
                        "GLD": 0.20
                        },
    "benchmark"       : "SPY",
    "startDate"       : "Dec 2010 04:00AM",
    "endDate"         : "Oct 2016 12:00AM",
    "strategy"        : "monthlyTradeStrategy_randomActionAgent",
    "equity"          : 50000.00,
    "status"          : "A",
    "alpha_params"    : [0.2, 0.5, 0.9],
    "gamma_params"    : [0.2,0.8],
    "epsilon_params"  : [0.2,0.8],
    "total_trials"    : 1
 },
```

5. Design

Our goal is to achieve maximum gain in a year and beat the market average. This framework aims to maximize the profits of investments by considering not only monthly trends of stock prices but also day price movements of stocks. I have instrumented weekly, monthly and quarterly strategies which will use price action based indicators coming from weekly, monthly and quarterly price change indicators.

There are 3 types of agents and each agent has its own goal to achieve and interacts with others to share episodes in the learning process:



5.1. Random Action Agent

Buy and SELL decision is performed by random action. In this project, we have used Random Action agent for 3 different strategies – weekly, monthly and quarterly. You can configure and change [add more strategies] this using instances, strategies.dict files in data/strategies directory

5.2. Random Q-Learning Agent

Buy and SELL decision is performed by using Q-table. However, the reward function uses random approach to decide the reward.

In this project, we have used Random Action agent for 3 different strategies – weekly, monthly and quarterly. You can configure and change [add more strategies] this using instances, strategies.dict files in data/strategies directory

5.3. Flexible Q-learning Agent

Buy and SELL decision is performed by Agent that uses reinforcement technique using Q-table. The reward function uses several indicators and associated signals to derive the appropriate reward.

The {state} variable used can have any number of signals based on the strategy used. I have developed flexible, dynamic way of adding more state member which then can instructed via the strategy class. With this approach, I can add a new strategy class which can opt to use as many indicators for state member and as many indicators for reward calculation.

States, Actions, and Rewards

One of the most important keys to achieve reasonable performance in machine learning is the representation of the input space.

The calculation of reward is as follows.

The agent is given zero reward for the first BUY signal calculation of its reward is postponed until the agent sells the stock (or sell signal is received)

The rate of daily changes of stock prices are given while the agent takes HOLD. But when it takes SELL, zero reward is given because the signal means that it exits the market.

When the sell price is determined by the agent, the buy signal agent receives the profit rate, considering the transaction cost (*transaction cost is not yet instrumented in my code yet),* as the reward.

Learning Algorithm

An episode starts with a stock at a certain day chosen by the environment (input configuration). If the agent takes NOT-BUY as a response to the state of the given stock, the episode ends and a new episode starts.

If the stock cannot be purchased, the episode ends and invokes the buy signal agent with 0 as the reward.

Each agent has equations for reward, delta and Q-value, but all of them are expressed in the same notation for the brevity. Time index is also

abbreviated for the same reason. If the agent fails to sell the stock with its offer price, it sells the stock at close, the last traded price of the day. The update rules and function approximation are based on the ordinary Q-learning.

TD Sequential vs conventional momentum indicators

Conventional momentum indicators such as RSI are typically calculated between 0 to 100 have constraint of overbought and oversold zones which makes them less reliable when price action switches between ranges and trends. TD Setup indicators, on the other hand adjust dynamically in line with the price action.

An Q-learning example with ε-greedy policy:

Use the Q-learning update function:

$Q(s,a)=Q(s,a)+\alpha*(R+\gamma*\max(Q(a,a')-Q(s,a))$
where R is defined as:
$R=r_t*state*wealth_t*|state-action|$
and r is the percentage return at time t.

Action:
{BUY, SELL, HOLD}

No of States in Q-Learning Strategy 1

- Moving average is used as a primary technique to decide BUY/SELL/HOLD.
- If the short window moving average > long window overage, then there is a possibility of a trend reversal
- This is one of the basic and well know technical indicators

Following are the field used to define the state

State = { prev_action, sw_gt_lw_moving_average_signal, position_change_per}

No of states in the state definition I have selected is

= No of positions *
[#of possible prev action] *
[#sw_gt_lw_moving_average_signals] *
[Position % increase (round to digit)

= No of positions * [BUY, SELL, NONE] * [YES, NO] * [1 … 100]

If we assume that No of positions for our experiment is around 10 for diversified EFT based portfolio, then total no of states is

= 10 * 3 * 2 100
= 6000

The total (max) number of states is 6000 and is reasonable to hold in a memory. Q-table data structure is compact and for each run look up time to get the Q info will be efficient.

No of States in Q-Learning Strategy 2

- In this strategy, two crossover signals are used. It is very similar to Q-Learning strategy 1 but the idea is to combine multiple indicators which helps in deciding the trade.

- The indicators used here could RSI, MACD and any other momentum indicators. I have so far developed only RSI based indicators (14_20 and 50_200) but one can more those indicators easily with this framework. Please note that when you have more indicators, of course, the State table will become huge.

State:

State = { prev_action, rsa_14_50_days_crossover_signal, rsa_14_50_days_crossover_signal, position_change_per }

No of states in the state definition I have selected is

= No of positions *
[#of possible prev action] *
[# rsa_14_50_days_crossover_signal] *
[# rsa_14_50_days_crossover_signal] *
[Position % increase (round to digit)

= No of positions * [BUY, SELL, NONE] * [YES, NO] * [YES,NO]* [1 … 100]

If we assume that No of positions for our experiment is around 10 for diversified EFT based portfolio, then total no of states is

= 10 * 3 * 2* 2 * 100
= 12000

No of States in Q-Learning Strategy 3

In this approach, new kind of indicator is added. It is called de-mark TD Sequence indicator.

State = { prev_action, rsa_14_50_days_crossover_signal, rsa_14_50_days_crossover_signal, *TD_seq_indicator*, position_change_per }

No of states in the state definition I have selected is

= No of positions *
[#of possible prev action] *
[# rsa_14_50_days_crossover_signal] *
[# rsa_14_50_days_crossover_signal] *
[#*TD_seq_indicator] **
[Position % increase (round to digit)

= No of positions * [BUY, SELL, NONE] * [YES, NO] * [YES, NO]* [YES, NO] * [1 … 100]

If we assume that No of positions for our experiment is around 10 for diversified EFT based portfolio, then total no of states is

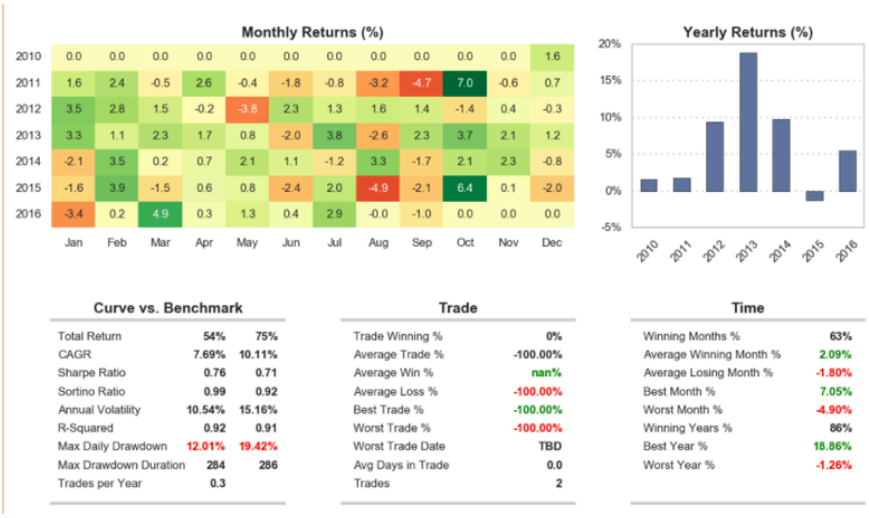
= 10 * 3 * 2* 2 * 2 *100
= 24000

6. Evaluation metrics

The objective of the above reinforcement algorithm based approach is to integrate various optimal policies and learn for maximum performance gain.

Each run will produce a report which includes

- Total Return,
- CAGR
- Sharp Ratio, Annual Volatility, R-Squared and Maximum Daily Draw Down



In our experiment, you will see several of these reports involving 3 different strategies trying 3 different learning algorithms with several possible combinations of alpha, gamma and epsilon. You can find the reports in data/reports direcotry

7. Input and running the algorithm

7.1. Indicators

- data/strategies/indicators.dict
- One can add new indicators under smartrader/indicator directory. For example, you will see rsi_indicators.py and demark-indicators.py
- You can add new indicators in these files or create a new file and add indicators
- The new indicators created should be defined in indicatros.dict for to get processed

```
{
    "rsi_indicators.RSI14" : {
        "indicator"         : "rsi_indicators.RSI14",
        "window"            : 14,
        "status"            : "A",
        "dependencies"      : []
    },
    "rsi_indicators.RSI50" : {
        "indicator"         : "rsi_indicators.RSI50",
        "window"            : 50,
        "status"            : "A",
        "dependencies"      : []
    },

    "rsi_indicators.RSICrossOver" : {
        "indicator"         : "rsi_indicators.RSICrossOver",
        "status"            : "A",
        "dependencies"      : ["rsi_indicators.RSI14", "rsi_indicators.RSI50"]
    },

    "rsi_indicators.DailyValueChangeCalculator" : {
        "indicator"         : "rsi_indicators.DailyValueChangeCalculator",
        "window"            : 50,
        "status"            : "A",
        "dependencies"      : []
    },

    "rsi_indicators.MonthlyValueChangeCalculator" : {
        "indicator"         : "rsi_indicators.MonthlyValueChangeCalculator",
        "window"            : 50,
        "status"            : "A",
        "dependencies"      : []
    },

    "rsi_indicators.QuaterlyValueChangeCalculator" : {
        "indicator"         : "rsi_indicators.QuaterlyValueChangeCalculator",
        "window"            : 50,
        "status"            : "A",
        "dependencies"      : []
    },

    "rsi_indicators.movingAverage" : {
        "indicator"         : "rsi_indicators.movingAverage",
        "window"            : 50,
        "status"            : "A",
```

7.2. Strategies

- data/strategies/strategies.dict
- One can add new strategy under smartrader/strategies directory. For example, you will see following strategies now.
    - monthlyTradeStrategy.py
    - quaterlyTradeStrategy.py
    - simpleBuyAndHold.py
    - weeklyTradeStrategy.py
- You can add new strategies in these files or create a new file and add strategies
- The new indicators created should be defined in strategies.dict for to get processed

```
"weeklyTradeStrategy_learningAgent1":{
   "positionSizer"           :"rebalance.LiquidateRebalancePositionSizer",
   "riskManager"             :"example.ExampleRiskManager",
   "portfolioHandler"        :"portfolio_handler.PortfolioHandler",
   "complianceHandler"       :"example.ExampleCompliance",
   "executionHandler"        :"ib_simulated.IBSimulatedExecutionHandler",
   "title"                   :"weeklyTradeStrategy_learningAgent1",
   "tearsheetStatistics"     :"tearsheet.TearsheetStatistics",
   "displayStrategy"         :"DisplayStrategy",
   "strategyHandler"         :"weeklyTradeStrategy",
   "learningHandler"         :"learningAgent1"
},
"monthlyTradeStrategy_learningAgent1":{
   "positionSizer"           :"rebalance.LiquidateRebalancePositionSizer",
   "riskManager"             :"example.ExampleRiskManager",
   "portfolioHandler"        :"portfolio_handler.PortfolioHandler",
   "complianceHandler"       :"example.ExampleCompliance",
   "executionHandler"        :"ib_simulated.IBSimulatedExecutionHandler",
   "title"                   :"monthlyTradeStrategy_learningAgent1",
   "tearsheetStatistics"     :"tearsheet.TearsheetStatistics",
   "displayStrategy"         :"DisplayStrategy",
   "strategyHandler"         :"monthlyTradeStrategy",
   "learningHandler"         :"learningAgent1"
},
"quaterlyTradeStrategy_learningAgent1":{
   "positionSizer"           :"rebalance.LiquidateRebalancePositionSizer",
   "riskManager"             :"example.ExampleRiskManager",
   "portfolioHandler"        :"portfolio_handler.PortfolioHandler",
   "complianceHandler"       :"example.ExampleCompliance",
   "executionHandler"        :"ib_simulated.IBSimulatedExecutionHandler",
   "title"                   :"quaterlyTradeStrategy_learningAgent1",
   "tearsheetStatistics"     :"tearsheet.TearsheetStatistics",
   "displayStrategy"         :"DisplayStrategy",
   "strategyHandler"         :"quaterlyTradeStrategy",
   "learningHandler"         :"learningAgent1"
},
```

7.3. Instances

- data/strategies/instances.dict
- instances configuration determines the scope of actual run.
- runMain and framework parses this and run the experiment for the instances and input data set.
- You can add any number of instances and the framework takes care of running all instances one by one

```
"instance4" : {
    "experiment"    : "experiment1",
    "tickerWeights" : {
                        "SPY": 0.7,
                        "GDX": 0.1,
                        "GLD": 0.2
                        },
    "benchmark"       : "SPY",
    "startDate"       : "Dec 2010 04:00AM",
    "endDate"         : "Oct 2016 12:00AM",
    "strategy"        : "weeklyTradeStrategy_learningAgent1",
    "equity"          : 50000.00,
    "status"          : "D",
    "alpha_params"    : [0.2],
    "gamma_params"    : [0.2],
    "epsilon_params"  : [0.2],
    "total_trials"    : 10
},

"instance5" : {
    "experiment"    : "experiment1",
    "tickerWeights" : {
                        "SPY": 0.20,
                        "IJS": 0.05,
                        "EFA": 0.15,
                        "EEM": 0.05,
                        "AGG": 0.20,
                        "JNK": 0.05,
                        "DJP": 0.10,
                        "GLD": 0.20
                        },
    "benchmark"       : "SPY",
    "startDate"       : "Dec 2010 04:00AM",
    "endDate"         : "Oct 2016 12:00AM",
    "strategy"        : "monthlyTradeStrategy_learningAgent1",
    "equity"          : 50000.00,
    "status"          : "A",
    "alpha_params"    : [0.2, 0.5, 0.9],
    "gamma_params"    : [0.2,0.8],
    "epsilon_params"  : [0.2,0.8],
    "total_trials"    : 10
},
```

8. Automation tools, logs and embedded runtime statistics

As part of this project. I have developed several automation scripts that enables one to run experiment with several different alpha,

gamma and epsilon values in a flexible way that can be initiated from command line. This automaton enables one to initiate an experiment with different trials, hyperparametes without changing the code.

Moreover, I have instrumented the code base (agent.py, environment.py) etc to record run time statistics about the trial and its progress. Up on completion of experiment, an analysis document is created (by analysis the results program) which summarizes the top trials/experiments having high rewards and completion rate.

9. Implementation & Code snippets

9.1. Framework

- Framework consist of the main program and many sub components
- Each sub component can be considered as a plugin
- All the plugins for each run is assembled in Agent Factory
- Framework takes of persisting the results in data/results directory
- Framework can be started by app/runMain.py

9.1.1. Agent Factory

- Reads the plugins and configuration and assemble/wire the plugin components
- As you see below, there are various plugins such as position_sizer, risk_manager, indicators, strategies, display, statistics etc.
- It is the responsibility of Agent Factor to assemble this based on strategy.dict and make it ready for the run

```
        tmpArray=strategyHandler.split(".")
        # Use the monthly liquidate and rebalance strategy
        strategyClass = getattr(importlib.import_module("smarttrader.strategy."+tmpArray[0]), "Strategy")
        displayClass  = getattr(importlib.import_module("smarttrader.strategy.display"), displayStrategy)

        self.strategy = strategyClass(self.tickers, self.events_queue)
        strategy = Strategies(self.strategy, displayClass())

        # Use the liquidate and rebalance position sizer
        # with prespecified ticker weights
        tmpArray = positionSizer.split(".")
        rebalancingClass = getattr(importlib.import_module("smarttrader.position_sizer."+tmpArray[0]), tmpArray[1])
        self.position_sizer = rebalancingClass(ticketWeights)


        # Use an example Risk Manager
        tmpArray = riskManager.split(".")
        riskManagerClass = getattr(importlib.import_module("smarttrader.risk_manager."+tmpArray[0]), tmpArray[1])
        self.risk_manager = riskManagerClass()

        # Use the default Portfolio Handler
        tmpArray = portfolioHandler.split(".")
        portHandlerClass = getattr(importlib.import_module("smarttrader.portfolio."+tmpArray[0]), tmpArray[1])
        self.portfolio_handler = portHandlerClass(self.initial_equity, self.events_queue, self.price_handler,
          self.position_sizer, self.risk_manager)

        # Use the ExampleCompliance component
        tmpArray = complianceHandler.split(".")
        complianceClass = getattr(importlib.import_module("smarttrader.compliance."+tmpArray[0]), tmpArray[1])
        self.compliance = complianceClass(config)

        # Use a Indicator component
        self.indicator_list = list()
        self.indicator_dict = dict()
        for indicator, value in indicators.iteritems():
            print indicator, value
            tmpArray = value['indicator'].split(".")
            indicatorClass = getattr(importlib.import_module("smarttrader.indicators."+tmpArray[0]), tmpArray[1])
            instance = indicatorClass(self.tickers, self.events_queue,value)
            self.indicator_list.append(instance)
            self.indicator_dict[value['indicator']] = instance


        # Use a simulated IB Execution Handler
```

9.2. Playback Simulator

- Very important part of the application
- It basically invokes the pricing/market data provider and for each data tick
- Starting from the start date of the trading, it calls the indicators, process the indicators for the tick and call appropriate strategy class.

```
print("Running Backtest...")
while self.price_handler.continue_backtest:
    try:
        event = self.events_queue.get(False)
    except queue.Empty:
        self.price_handler.stream_next()
    else:
        if event is not None:

            print ("------------------ Price EVENT Received from market backtest----------------------")
            if event.type == EventType.TICK or event.type == EventType.BAR:
                self.cur_time = event.time

                graph = dict()
                indicatorPath = getPath('strategies.data.path')
                indicators = json.load(open(indicatorPath + "indicators.dict"))
                for indicator, value in indicators.iteritems():
                    dependencies = value['dependencies']
                    graph[indicator] = dependencies

                order = topoSort.topological(graph)
                while True:
                    try:
                        ind = order.pop()
                        print (self.indicator_dict)
                        indClass = self.indicator_dict[ind]
                        indClass.updateIndicators(event, self.agent)

                    except IndexError:
                        break

                print("        ***** Processing signals  ----------------------")
                self.strategy.calculate_signals(event,self.agent)
                print("        ***** Update portfolio  ----------------------")
                self.portfolio_handler.update_portfolio_value()
                print("        ***** Update Stats  ----------------------")
                self.statistics.update(event.time, self.portfolio_handler)

            elif event.type == EventType.SIGNAL:
                self.portfolio_handler.on_signal(event)
            elif event.type == EventType.ORDER:
                self.execution_handler.execute_order(event)
            elif event.type == EventType.FILL:
                self.portfolio_handler.on_fill(event)
```

9.2.1. Indicators

Framework invokes Indicator classes for two reasons. First one is to process the market data and identify the signals (such as SELL or BUY). Second one is to provide the signal up on the request.

```
        self.invested = False

        self.sw_bars = dict()
        self.lw_bars = dict()
        self.signal = dict()

        self.tickers = tickers
        self.events_queue = events_queue
        self.ticks = 0

        self.period = indicator_details['window']

        for ticker in tickers :
            self.sw_bars[ticker] = deque(maxlen=self.short_window)
            self.lw_bars[ticker] = deque(maxlen=self.long_window)

        for ticker in tickers :
         self.signal[ticker] = "NONE"


    def getSignal(self, ticker):
        print "TIC", ticker, self.signal
        return SignalEvent(ticker, self.signal[ticker])

    def updateIndicators(self, event, agent=None):

        if event.type in [EventType.BAR, EventType.TICK] and event.ticker in self.tickers :

            self.lw_bars[event.ticker].append(event.adj_close_price)
            self.sw_bars[event.ticker].append(event.adj_close_price)


            # Enough bars are present for trading
            if len(self.lw_bars[event.ticker]) == self.lw_bars[event.ticker].maxlen:
                # Calculate the simple moving averages
                short_sma = np.mean(self.sw_bars[event.ticker])
                long_sma = np.mean(self.lw_bars[event.ticker])
                # Trading signals based on moving average cross
                if short_sma > long_sma and not self.invested:
                    print("LONG: %s" % event.time)
                    self.signal[event.ticker] = "BOT"

                elif short_sma < long_sma and self.invested:
                    print("SHORT: %s" % event.time)
```

9.3. Agents

9.3.1. Random Action Agent

```
class Action():

    def __init__(self, context,instanceDict ):
        self.state = State()

    def reset(self, destination=None):
        print "Inside reset"

    def getTotalRewards(self):
        return 0

    def getLegalActions(self):
        """
        returns the legal action from the current state
        """
        return ['BOT', 'SLD', 'NONE']

    def update(self, event,  learning_indictors, reward_indicators,indicator_dict, portfolio):

        pri Parameter 'learning_indictors' value is not used more... (⌘F1) _dict

        # Select action according to your policy
        action = random.choice(list(self.getLegalActions()))

        try:
            print " Action ", action
            if DEBUG:
                input("")

        except SyntaxError:
            pass


        return action
```

9.3.2. Simple/Random Q-Learning Agent

```
def getTotalRewards(self):
    return self.totalRewards

def act(self,event, action,reward_indicators, indicator_dict):
    # random reward. You would see different reward structure based on portfolio/position increase in learningAgent1
    return random.choice(list([0.5, 0.8, 2, 1, 0.2, 0.7, 2, 3]))
```

9.3.3. Flexible Q-learning Agent

```
class State() :

    def __init__(self, ticker, signals=None, lastAction=None) :
        self.signals = signals
        self.ticker = ticker
        self.lastAction = lastAction

    def getStateString(self):

        print "Inside learningAgent1:getStateString()"
        returnStr = self.ticker + ":"
        for key, value in self.signals.iteritems():
            returnStr += "'" + str(key) + ":" + str(value.action)
        returnStr += "--" + self.lastAction
        return returnStr
```

```
71
72      def update(self, event,  learning_indictors, reward_indicators,indicator_dict, portfolio):
73
74          print "Inside learningAgent1:Action:Update:", event, indicator_dict
75
76          signals = dict()
77          for ln in learning_indictors :
78              signals[ln] = indicator_dict[ln].getSignal(event.ticker)
79
80          # Update state from env. In this case, backtest/replay will call the update function
81          tempState = State(event.ticker, signals,"NONE")
82          print "current STATE:", tempState.getStateString()
83
84          # Select action according to your policy
85          action = self.qTable.chooseAction(tempState)
86
87          # Execute action and get reward
88          reward = self.act(event, action, reward_indicators, indicator_dict)
89
90          try:
91              print " Action ", action
92              print " Reward for action ", reward
93              if DEBUG:
94                  input("")
95
96          except SyntaxError:
97              pass
98
99          newstate = State(event.ticker,signals,action)
100
101         # Learn policy based on state, action, reward
102         self.qTable.updateQTable1(self.state,
103                                   action=action,
104                                   reward=reward,
105                                   newstate=newstate)
106         self.state = newstate
107         self.totalRewards += reward
108
109         return action
110
```

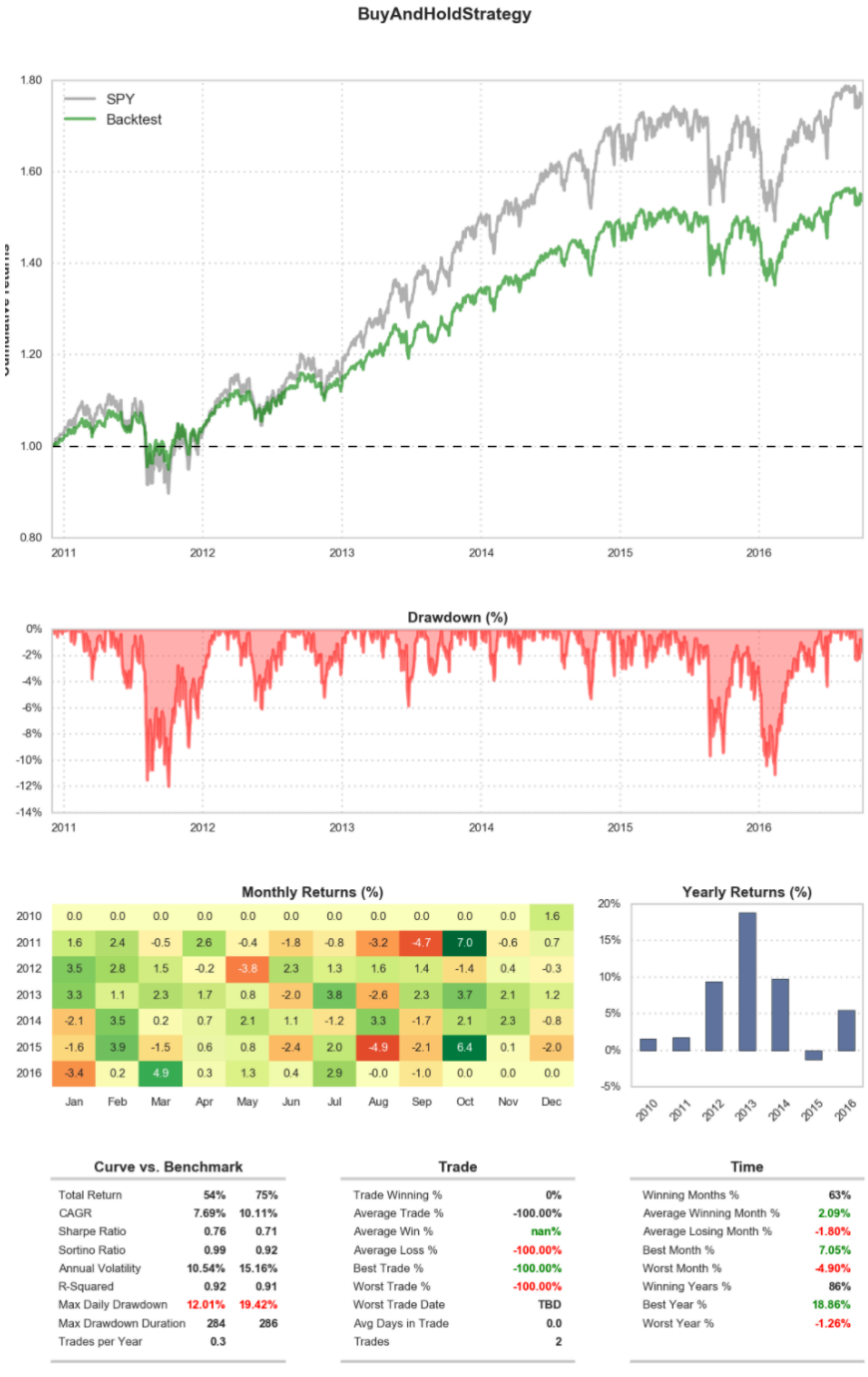
```
def getTotalRewards(self):
    return self.totalRewards

def act(self,event, action,reward_indicators, indicator_dict):

    # As of now, only one reward indicator can be passed.
    value = indicator_dict[reward_indicators[0]].getSignal(event.ticker)
    print "% chaned value:", value

    ## Since, I am passing weekly performance/value change as reward indicator
    ## the change is not expected to be more than 10%. However, if I start using monthlty or Qty reward indicator
    ## the value likely to be higher. In that case, we need to normalize.  The idea here is tha
    ## if you get 10% up in a week, the reward given is maximum -which is 1

    if value > 10:
        return 1
    elif value > 5 :
        return 0.5
    elif value > 2 :
        return 0.1
```

```
163
164     def chooseAction(self, state):
165         print "\n"
166         if True or random.random() < self._epsilon:
167             legalActions = self.getLegalActions()
168             bestAction = random.choice(legalActions)
169             print " Going for random action :", bestAction
170             return bestAction
171         else:
172
173             maxQ = 0.0
174             bestAction = "NONE"
175             for action in self.getLegalActions() :
176                     q = self.getQValue(state,action=action)
177                     if q >= maxQ :
178                         bestAction = action
179                         maxQ = q
180                     print " Going for best action for the state:", state.getStateString() , "=>", action,"\t\t",  q, ":", \
181                         maxQ
182
183
184             if maxQ == 0.0  :
185                 legalActions = self.getLegalActions()
186                 bestAction = random.choice(legalActions)
187                 print " No prev state-action found. Going for random action for the state:", bestAction
188
189             else :
190                 print " ++++++++++++++++++++ Self Learned action for the state:", bestAction, "having Q:", maxQ
191
192             return bestAction
193
194
195     def updateQTable1(self,state,action,reward,newstate):
196
197         oldQvalue = self.getQValue(state,action=action)
198         maxValue = self.getMaxQValue(state)
199
200         newQvalue = oldQvalue  + self._alpha * ( (reward + self._gamma * maxValue) - oldQvalue)
201         self.setQValue(state,action=action,new_value=newQvalue)
202
203         print " QTable  : "
204         print json.dumps(self._table, indent=4)
```

10. Run & Observation

10.1. Random Action Agent

Weekly Rebalancing

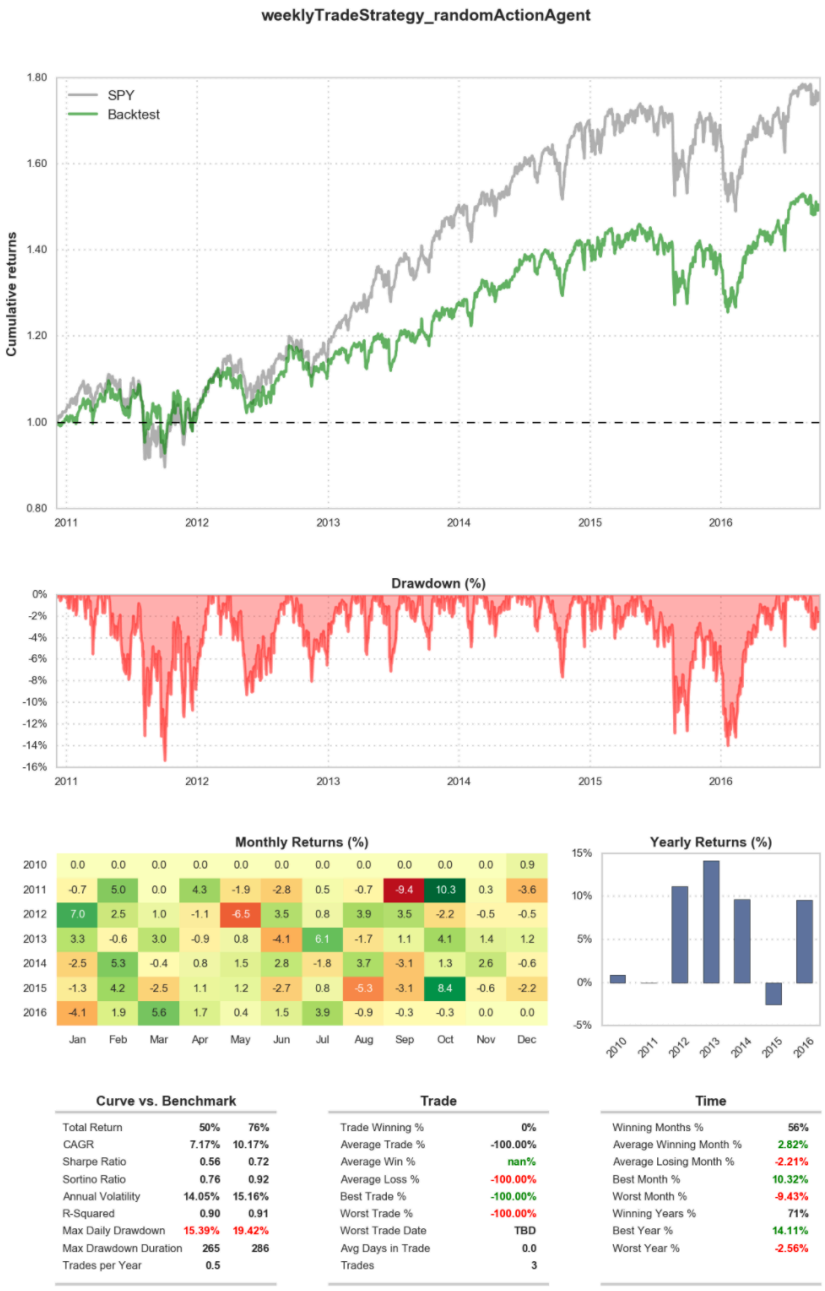


Curve vs. Benchmark		
Total Return	54%	75%
CAGR	7.69%	10.11%
Sharpe Ratio	0.76	0.71
Sortino Ratio	0.99	0.92
Annual Volatility	10.54%	15.16%
R-Squared	0.92	0.91
Max Daily Drawdown	12.01%	19.42%
Max Drawdown Duration	284	286
Trades per Year	0.3	

Trade	
Trade Winning %	0%
Average Trade %	-100.00%
Average Win %	nan%
Average Loss %	-100.00%
Best Trade %	-100.00%
Worst Trade %	-100.00%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	2

Time	
Winning Months %	63%
Average Winning Month %	2.09%
Average Losing Month %	-1.80%
Best Month %	7.05%
Worst Month %	-4.90%
Winning Years %	86%
Best Year %	18.86%
Worst Year %	-1.26%

Monthly Rebalancing



Quarterly rebalancing

Not included in the word doc to reduce space. Pease refer data/results if you want to see the details

10.2. Simple/Random Reward Q-Learning Agent

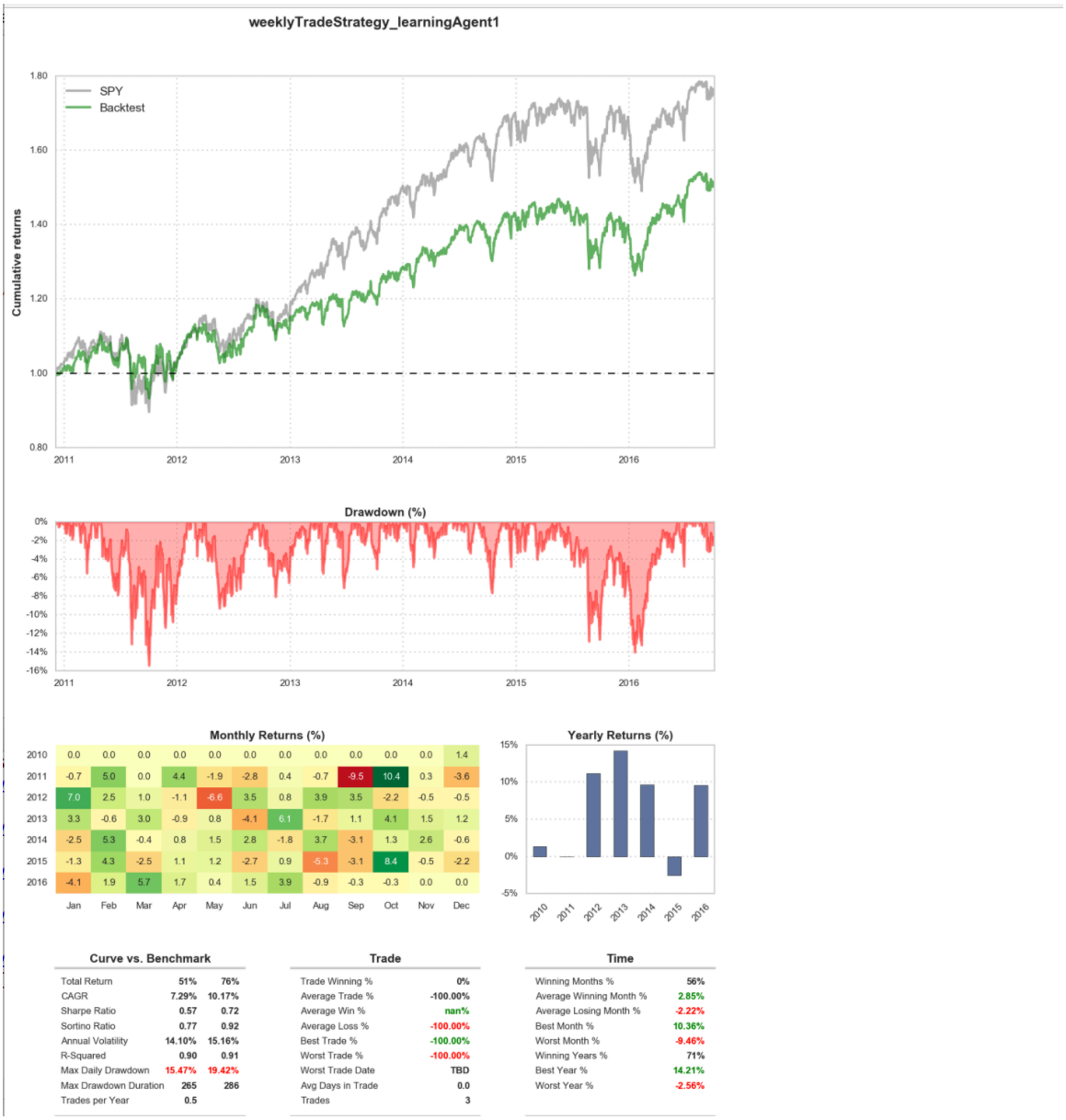
Not included in the word doc to reduce space. Pease refer data/results if you want to see the details

10.3. Flexible Q-learning Agent

This is the most interesting experiment. First, I ran with low alpha, gamma and Epsilon for many trials. In most cases, I could not beat the market.

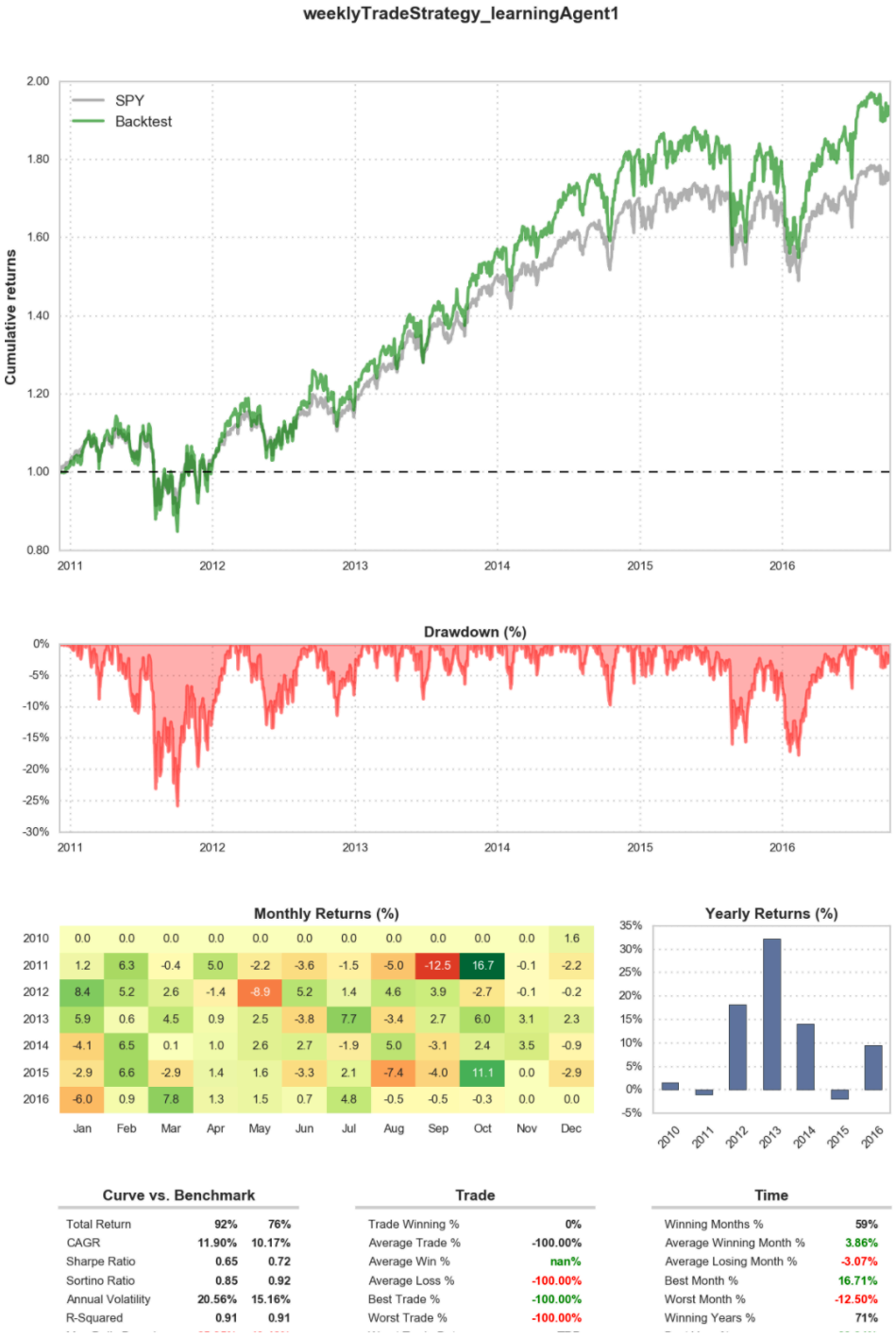
Weekly Rebalancing

Alpha 0.2, Gamma, 0.2 Epsilon 0.2



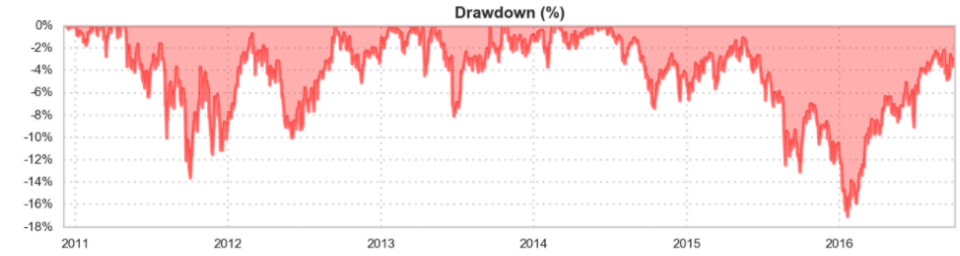
I observed that as I increase Gamma and Epsilon, the results started to looking better. After many trials, following is the best performance I observed.

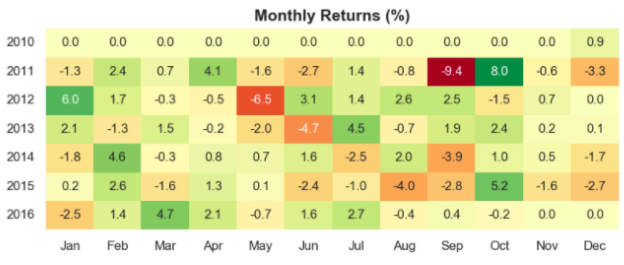
Alpha 0.5, Gamma, 0.2 Epsilon 0.2 and Iteration 10

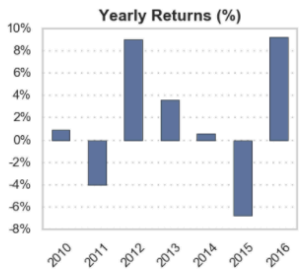


Monthly Rebalancing

Monthly rebalancing is not helping much. I am yet to try with different params.







Curve vs. Benchmark		
Total Return	12%	76%
CAGR	1.99%	10.17%
Sharpe Ratio	0.24	0.72
Sortino Ratio	0.32	0.92
Annual Volatility	10.66%	15.16%
R-Squared	0.34	0.91

Trade	
Trade Winning %	0%
Average Trade %	-100.00%
Average Win %	nan%
Average Loss %	-100.00%
Best Trade %	-100.00%
Worst Trade %	-100.00%

Time	
Winning Months %	55%
Average Winning Month %	2.09%
Average Losing Month %	-2.11%
Best Month %	8.00%
Worst Month %	-9.43%
Winning Years %	71%

Quarterly rebalancing

11. Improve the Q-Learning Driving Agent

- Enhance Q-learning agent so that, after sufficient training, the agent is able to beat the benchmark or perform better.

- Parameters in the Q-Learning algorithm, such as the learning rate (alpha), the discount factor (gamma) and the exploration rate (epsilon) all contribute to the agent's ability to learn the best action for each state.

You can specify the params in instances.dict file. For example, for following instance, different values of alpha, gamma and epsilon is provided

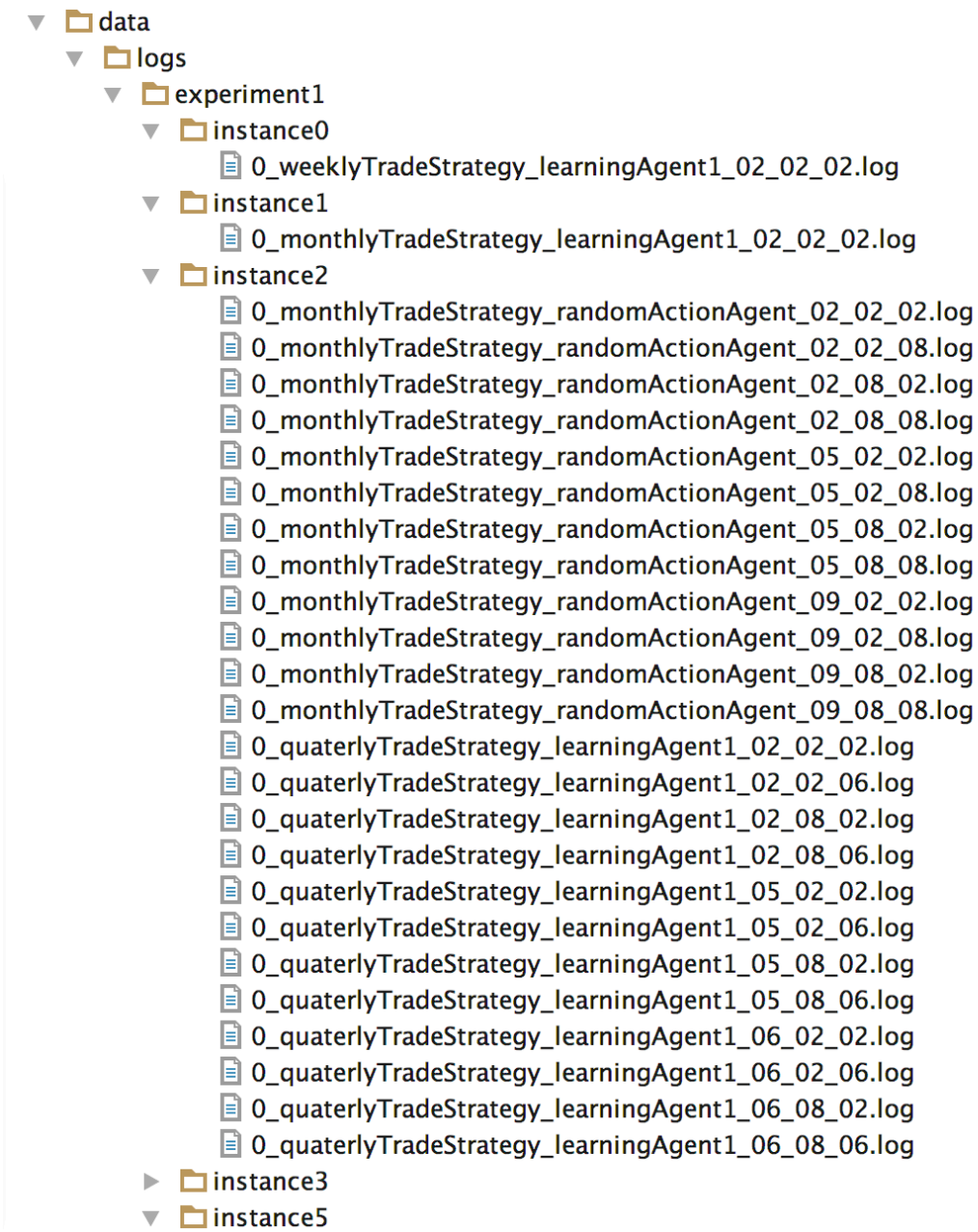
```
"instance5" : {
      "experiment"    : "experiment1",
      "tickerWeights" : {
                          "SPY": 0.20,
                          "IJS": 0.05,
                          "EFA": 0.15,
                          "EEM": 0.05,
                          "AGG": 0.20,
                          "JNK": 0.05,
                          "DJP": 0.10,
                          "GLD": 0.20
                          },
      "benchmark"       : "SPY",
      "startDate"       : "Dec 2010 04:00AM",
      "endDate"         : "Oct 2016 12:00AM",
      "strategy"        : "monthlyTradeStrategy_learningAgent1",
      "equity"          : 50000.00,
      "status"          : "A",
      "alpha_params"    : [0.2, 0.5, 0.9],
      "gamma_params"    : [0.2,0.8],
      "epsilon_params"  : [0.2,0.8],
      "total_trials"    : 10
   },
```

Run the simulation. Adjust one or several of the above parameters and iterate this process.

With automation tools, logs and runtime stats and analysis over runtime stats, we will run two experiments. Each experiments consist of several trials of various combination of alpha, gamma and epsilon.

Now, when you run mainDriver.py it will automatically run for various combination of alpha, gamma and epsilon from the above sets.

For each run, the corresponding log file is created under data/logs

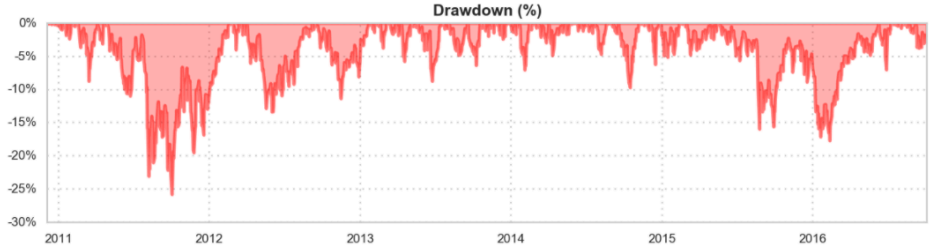


After completing the experiment, grep on Total rewards under logs directory.

Best RUN from many trials…

Alpha 0.5, Gamma, 0.2, Epsilon 0.2 and Iteration 10

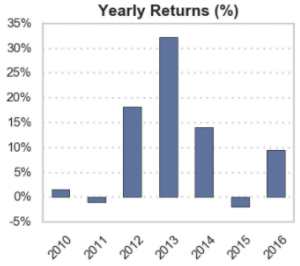
weeklyTradeStrategy_learningAgent1



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2010	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.6
2011	1.2	6.3	-0.4	5.0	-2.2	-3.6	-1.5	-5.0	-12.5	16.7	-0.1	-2.2
2012	8.4	5.2	2.6	-1.4	-8.9	5.2	1.4	4.6	3.9	-2.7	-0.1	-0.2
2013	5.9	0.6	4.5	0.9	2.5	-3.8	7.7	-3.4	2.7	6.0	3.1	2.3
2014	-4.1	6.5	0.1	1.0	2.6	2.7	-1.9	5.0	-3.1	2.4	3.5	-0.9
2015	-2.9	6.6	-2.9	1.4	1.6	-3.3	2.1	-7.4	-4.0	11.1	0.0	-2.9
2016	-6.0	0.9	7.8	1.3	1.5	0.7	4.8	-0.5	-0.5	-0.3	0.0	0.0

Yearly Returns (%)



Curve vs. Benchmark

Total Return	92%	76%
CAGR	11.90%	10.17%
Sharpe Ratio	0.65	0.72
Sortino Ratio	0.85	0.92
Annual Volatility	20.56%	15.16%
R-Squared	0.91	0.91

Trade

Trade Winning %	0%
Average Trade %	-100.00%
Average Win %	nan%
Average Loss %	-100.00%
Best Trade %	-100.00%
Worst Trade %	-100.00%

Time

Winning Months %	59%
Average Winning Month %	3.86%
Average Losing Month %	-3.07%
Best Month %	16.71%
Worst Month %	-12.50%
Winning Years %	71%

12. Conclusion

By looking at experment1 logs results, we can realize that

- Beating stock market is not an easy task
- There are some strategies which beats the market. However please not that we have not considered transaction costs/tax etc.

- In my experiment, weekly rebalancing with multiple indicators (RSI, d-mark TD as signal indicator and weekly performance as reward indicator) with alpha=0.5, gamma=0.2 and epsilon=0.2 give the optimal result.

- Agent with alpha as 0.5 or 0.9 , gamma as 0.2 and epsilon as 0.2 runs with high reward and best completion rate with minimal penalty and best run time.
- Higher alpha means that agent uses higher learning rate. Learning rate controls the learning step size, that is, how fast learning takes place.
- In our experiment, higher alpha does not always result in best completion runs. Seems like 0.5 alpha is optimal. You can review the detail logs to confirm this.
- Future reinforcements are weights controlled by a value gamma between 0 and 1. A higher value of gamma means that the future matters more for the Q-value of a given action in a given state.
- In our experiment, higher gamma does not help much. Gamma with 0.2 value provides the best results.
- The reason for higher gamma is not helpful because the information calculated for reward may be faulty for one reason or another.
- It is better to update more gradually, to use the new information to move in a particular direction, but not to make too strong a commitment.
- Higher epsilon is counterproductive in our experiment. Usually higher epsilon helps when three is more randomness in the behavior of env response.
- Seems like, there is not much of variation of responses from env and

higher epsilon does not help much – hence low epsilon results in better success rate.

Behavior of Agents and reaching the optimal policy

- *Agent with alpha as 0.5 or 0.9 , gamma as 0.2 ad epsilon as 0.2 runs with high reward and best completion rate with minimal penalty and best run time.*
- The optimal policy for this kind of agent is to balance between exploration and exploitation. Alpha with 0.5 and gamma 0.2 achieves the optimal exploitation and 0.2 epsilon is optimal for exploration.

In summary, further study is needed to confirm the results – specifically across multiple bull and bear market.

With this framework, I am planning to further try different indicators and combinations and see whether we can consistently beat the market with a margin that will include transaction costs/tax in beating the market.

13. References

1. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint* arXiv:1312.5602 (2013).

2. http://cs229.stanford.edu/proj2014/David%20Montague,%20Algorithmic%20Trading%20of%20Futures%20via%20Machine%20Learning.pdf

3. Algorithmic and High-Frequency Trading, **Print ISBN-13:** 978-1-107-09114-6 **By:** Álvaro Cartea; Sebastian Jaimungal; José Penalva
   a. **Publisher:** Cambridge University Press

4. Analysis of Financial Time Series
   b. **By:** RUEY S. TSAY

c. **Publisher:** John Wiley & Sons
d. **Pub. Date:** August 30, 2010
e. **Print ISBN:** 978-0-470-41435-4
f. **Web ISBN:** 0-470414-35-9

5. Market-Neutral Trading: Combining Technical and Fundamental Analysis Into 7 Long-Short Trading Systems

g. **By:** Thomas K. Carr
h. **Publisher:** McGraw-Hill
i. **Pub. Date:** December 17, 2013
j. **Print ISBN-13:** 978-0-07-181310-5

6. quantstart.com for python libraries for yahoo data reader and portfolio management