Simulation-Assisted Policy Learning for American Options*

Jeremy Manuel

Mechanical & Industrial Engineering

University of Toronto

April 19th, 2020

Abstract

American call options are unique financial derivatives that give the owner the right, but not the obligation, to buy the underlying asset at *any point* during the option's lifetime. This results in a unique payoff dynamic between the option holder's decision making and the underlying assets price. Developing lucrative exercise policies for these options is a difficult, yet well defined financial engineering problem that is typically approached with dynamic programming. The purpose of this research is to investigate the application of reinforcement learning and stochastic simulation in learning effective exercise policies. This study provides an empirical, out-of-sample performance of learned policies on historical equity prices. Results indicate that significant economic gains can be made on short term option contracts using this portfolio of methodologies.

Key words: Reinforcement Learning, Stochastic Simulation, Geometric Brownian Motion, Proximal Policy Optimization, Markov Decision Process, Financial Engineering

*This study benefits from the research completed by Yuxi Li, Csaba Szepesvari and Dale Schuurmans in 'Learning Exercise Policies for American Options'. The respective authors employ least-squares policy iteration and fitted Q-iteration for learning exercise policies. Analysis of this problem is further explored with modern optimization algorithms, namely Proximal Policy Optimization.

Li, Yuxi. Szepesvari, Csaba. Schuurmans, Dale. (2009) *Learning Exercise Policies for American Options*. University of Alberta.

1 Introduction

Call options are financial contracts that give the owner the right, but not the obligation, to buy an asset at an agreed upon price at a future date. *American options* provide the holder the freedom to exercise this right at any point during the term of the contract. As a consequence, these contracts tend to trade at a higher premium[1]. However, they also provide a considerable advantage to an investor with economic intuition behind the underlying assets behaviour.

1.1 Problem Description

An option holder's 'policy' for determining exercise timing breaks down into an optimal stopping problem between the inception of the contract and the contract's expiration date. In general, a policy is defined as follows: an algorithm that generates a *signal* based on historical data - either technical or fundamental in nature - which in turn recommends a sequence of actions. A policy's effectiveness can thus be evaluated based on the result of its deployment under real market constraints

To support the evaluation criteria, a 'fair' option price is also defined. In theory, this fair price is a direct measurement of the optimal exercise policy governing the option. With perfect information available to every investor, all American options would be priced this way as a result of *market arbitrage*. In practice, this fair price is indeterminable, as each investor is guided by their own biases, belief systems and presuppositions. The discrepancy in asset evaluations is what allows for these contracts to be traded on an exchange. For the purposes of this analysis, a fair price will be determined by Monte Carlo experiments (Sec. 2.4).

1.2 Approach

Reinforcement learning is an intuitive approach to the described problem. Unlike the supervised and unsupervised machine learning paradigms, reinforcement learning involves training an *agent* in some unknown environment to produce a desirable outcome. This environment, which is mathematically defined as a Markov Decision Process, evaluates the agents *actions* and *states* with continuous feedback in the form of *rewards*. By maximizing cumulative reward, the agent can determine an optimal policy for accomplishing the desired objective. The optimal stopping problem for an American option fits trivially into this framework (Sec. 2.1)

One issue that arises quite often in machine learning is data scarcity. Deep learning in particular requires large data sets to develop complex and sophisticated models, especially in the presence of *nonlinearities*[2]. When data is limited (or, equivalently, lacks variation) the quality of learning is equally suppressed. Unfortunately, financial data quite often succumbs to this handicap. An equity, for example can at best be characterized by its longevity on the marketplace. Furthermore, the window of relevant data may prove to be even smaller.

Stochastic simulation is the primary tool used to address this issue. With parameters derived from archived data, asset paths are simulated with Geometric Brownian motion and used as a learning environment. Once trained, the RL agent then interacts with the historical asset path to test its performance. Policy performance is measured by an out of sample α (Sec 2.5) over a multitude of option scenarios. These empirical findings will be used to address the following research questions:

1. Can an effective exercise policy be learned with simulated data?

2. To what degree is this approach suitable for American options trading?

It is important to note that data availability is not the only concern regarding machine learning applications to finance. Low *signal to noise* ratio in financial timeseries forecasting is a well documented problem[3] that is difficult to remedy with learning algorithms. More so, it is unlikely that a stochastic process will reinvent this behaviour in a way that is more tractable for machine learning methodologies.

[1] The price at which a put/call option contract is sold/purchased. For the purpose of this analysis, this is synonymous with the options 'fair price'

[2] Bryan et al. (2019) *Empirical Asset Pricing via Machine Learning*, University of Chicago. Section 1.5.

[3] Gudelek et al. (2019) *Financial Time Series Forecasting with Deep Learning : A Systematic Literature Review* University of Economics and Technology, Ankara, Turkey.

Specifics: To simplify this analysis (and keep compute time reasonable) only Apple stock (AAPL) as an underlying asset is considered, with daily closing prices (CLOSE) as a univariate timeseries. Furthermore, only at-the-money[4] call options are considered. Otherwise, the empirical analysis extends to three different option lengths: one month contracts, quarterly contracts and annual contracts. Different variants of experiment design (observable variables, number of simulations, etc) are to be explored as well.

2 Methodology

This section lends itself to describing the chosen methods of analysis and experimental execution of these methods.

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a five tuple object composed of the following:

- A state space $\mathcal{S}$
- An action space $\mathcal{A}$
- A set of transition probability distributions $\mathcal{P}(\cdot|\mathcal{S}, \mathcal{A})$
- A reward function $\mathcal{R}$
- A discount factor γ

In general, events within an MDP proceed as follows: given a state s_t, an *agent* chooses an action a_t which maps them to some future state s_{t+1}. This mapping can either be stochastic or deterministic in nature (in the deterministic case, $\mathcal{P}(\mathcal{S}_{t+1}|\mathcal{S}_t, \mathcal{A}_t) = 1$). Rewards are generated based on the sequencing of states and actions the agent develops through the means of a *policy* π. The cumulative expected return or *goal* can be defined as

$$E_\pi(\mathcal{G}) = \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \quad \forall t$$

where rewards are temporally discounted by $0 < \gamma < 1$. An optimal policy π^* is defined as a policy that maximizes the expected cumulative return with optimal action and state sequencing.

[4] An option variant where the strike price is equal to the initial price.

An MDP serves as a framework through which policies can be optimized to achieve a desired objective. Reinforcement learning algorithms are the mechanisms that achieve this optimization with iterative based techniques.

The MDP relating to this analysis will now be defined. Consider an American option with maturity T, in days. Let S_t be the underlying assets price at time $t \leq T$, and K be the options strike price S_0. Suppose the choice to exercise the option is given by A_1 and the choice not to exercise is given by A_0. If a risk free rate r is assumed, then

- $\mathcal{S} = \{S_t\}, \quad 0 \leq t \leq T$
- $\mathcal{A} = \{A_i\}, \quad i = 0, 1$
- $\mathcal{R} = \begin{cases} R(S_t, A_i) = 0 & i = 0 \\ R(S_t, A_i) = max(S_t - K, 0) & i = 1 \end{cases}$
- $\gamma = e^{-\frac{r}{T}}$

Because the underlying asset is a random variable, the state space $\mathcal{S}$ can also be interpreted as a *continuous time stochastic process* indexed by $t \leq T$. While the generation of asset prices is stochastic, the transition probabilities are still considered deterministic (i.e, the transition of one randomly generated variable to another is deterministic given A_i). When the option is exercised, the agent enters an exit state and repeats the process.

2.2 Geometric Brownian Motion

In order to develop a training environment, the underlying equity needs to be simulated by some stochastic process. An intuitive choice as suggested by the literature is Geometric Brownian Motion (GBM), which is based on the natural process physics underlying an assets value (Brownian motion) and its strictly positive value (Geometric). A stochastic process S_t is said to follow GBM if it satisfies the stochastic different equation,

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

where 'percentage drift' μ and 'implied volatility' σ are constants, and W_t is a Brownian motion stochastic process. For an arbitrary value S_0, a closed form solution for S_t can be derived[5] using *Itô's Lemma*,

$$S_t = S_0 exp((\mu - \frac{\sigma^2}{2})t + \sigma W_t)$$

[5] Hull, John C. (2008). *Options, Futures and Other Derivatives* 7 edition, Section 14.6

For the purposes of simulating sample paths, S_t is discretized over the length of the options contract,

$$S_t = S_{t-1} exp((\mu - \frac{\sigma^2}{2})\frac{1}{T} + \sigma\sqrt{\frac{\epsilon}{T}})$$

with ϵ is i.i.d from $N(0,1)$

Under this discretized version, a normal distribution is fit to the log returns of S to estimate the parameters μ and σ. More specifically, the underlying transformation

$$R_t = ln(\frac{S_t}{S_{t-1}})$$

is fit using the formulas provided by maximum likelihood estimation,

$$\bar{u} = \frac{1}{T}\sum_{t=0}^{T-1} R_t, \quad \bar{v} = \sqrt{\frac{1}{T-1}\sum_{t=0}^{T-1}(R_t - \hat{u})^2}$$

These unbiased estimates are analogous to the drift and volatility terms used in generating sample paths[6] , i.e

$$\mu = \bar{u}T + \frac{1}{2}\sigma^2, \quad \sigma = \bar{v}\sqrt{T}$$

where the individual returns discretize some interval of time T. For non-dividend paying stocks, it is also common to use the risk free rate r as percentage drift.

2.3 Proximal Policy Optimization

The optimization algorithm that is used to parameterize π is, within the scope of this analysis, not elementary. As such, much of the mathematics are left out. A more rigorous description can be found in the publication from the original authors[7].

Optimizing a policy π is a reinforcement learning problem that splits itself into two camps - value based learning, which attempts to maximize expected return from action-value estimates, and policy gradient learning which directly parameterizes π, typically with a neural network. The latter tends to be more efficient in high dimensional environments and demonstrates better convergence. However, it still requires many samples to do so.

An improvement on this is to instead run two neural networks in parallel - one to update the policy π and one to evaluate the policy's actions (commonly referred to as an Actor-Critic approach). Actor-Critic methods are typically augmented with *advantage functions* that compare an action's value to the expected action at that state. If the action appears advantageous, the policy gradient is pushed in that direction (and vice versa).Consider a policy gradient estimator $\hat{g}$ with advantage function $\hat{A}_t$. Define an objective function with gradient $\hat{g}$ as

$$L_{PG}(\theta) = \hat{E}_t[log(\pi_\theta(a_t|s_t))\hat{A}_t]$$

Step sizes between new and old policies tend to be small so as to not collapse the policy's performance - which, as a consequence, limits convergence speed. Trust Region Policy Optimization (TRPO) introduces the idea of taking large step sizes constrained under a uniquely defined region called the 'trust region'. This constraint is roughly expressed in terms of *KL-Divergence*, a probability distribution measurement for comparing old and new policies. Even with a general approximation of this constraint, TRPO tends to avoid bad policy updates and improves upon general performance. Consider a KL penalized objective function that maximizes update size subject to some divergence target δ,

$$L_{KL}(\theta) = \hat{E}_t\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{old}}, \pi_\theta]\right)$$

where β is a *hyperparameter* that can be tuned. This schematic tends to work well even when the divergence is significantly different than δ.

Enter Proximal Policy Optimization (PPO), a family of policy gradient methods that employs benefits from both Actor-Critic and TRPO. Like Actor-Critic, PPO runs an old policy $\pi_{\theta_{old}}$ in an environment and computes $\hat{A}_t$ for each $t \in T$. It then optimizes $L_{KL}(\theta)$ using stochastic gradient descent and updates the old policy after the last epoch. PPO can also utilize other loss function variants and achieve similar rates of convergence. Its performance is state-of-the-art and has many well documented successes such as its implementation in Atari games.

[6] Dmouj, A. (2006) *Stock Price Modelling: Theory and Practice* Vrije University, Amsterdam

[7] Shulman et al (2017) *Proximal Policy Optimization Algorithms* OpenAI.

2.4 Experiment Design

In the empirical analysis, simulation serves two functions. The first is to develop a learning environment that allows a policy to *generalize* appropriately to historical data. The degree to which this is achieved can be evaluated by a trained policy's out-of-sample performance relative to some market benchmark (Sec 2.5).

Secondly, the 'fair price' of each option is evaluated using Monte Carlo simulations. This will be the option premium and is used in calculating the out-of-sample *time value* given the *intrinsic value*[8] an exercise policy generates. In this case, simulation error can be calculated directly by comparing the estimated premium with historical data. Given that financial data is noisy, Monte Carlo estimations can lack the necessary predictive power to evaluate future options accurately - which is why policies are compared against a market benchmark instead.

Simulation experiments are designed as follows: given a financial time series $\{X\}_N$ of daily closing prices, a set of κ artificial options, with arbitrary length ℓ, can be determined. These options are considered "artificial" because they're created from hypothetical scenarios and not real contracts. If $\{X\}_N$ is divided chronologically with a sliding window equal to ℓ, then the set of artificial options can be defined as

$$\Theta_\ell = \{x_1, x_2, ...x_\kappa\}, \quad \kappa = \lfloor \frac{N}{\ell} \rfloor$$

where

$$x_k = [X_{\ell k+1}, X_{\ell k+2}, ...X_{\ell k+\ell}], \quad 1 \leq k \leq \kappa$$

A major consideration in this design is compute time. By limiting the number of scenarios, intra scenario replication sizes can still be quite large while keeping computations reasonable.

Each scenario is estimated with a *simulation experiment* $\hat{x}_k(\ell, n, U)$, where n is the number of simulation replications and U is an arbitrary random number stream. Each replication $\hat{x}_k^{(i)}$ is one realization of a continuous-time stochastic process indexed by length $T = \ell$. As mentioned prior, the stochastic process proposed is a Geometric Brownian motion with parameters μ_k and σ_k estimated under log-normally distributed returns. Then, each replication is computed the following way:

$$\hat{x}_k^{(i)} = [S_{k,0}^{(i)}, S_{k,1}^{(i)}, ...S_{k,\ell}^{(i)}] \quad (S_{k,0} = X_{\ell k} \quad \forall i \in n)$$

where

$$S_{k,t}^{(i)} = S_{k,t-1}^{(i)} exp((\mu_k - \frac{\sigma_k^2}{2})\frac{t}{\ell} + \sigma_k \sqrt{\frac{\epsilon}{\ell}})$$

and ϵ is a standard normal random variable generated by U.

Reinforcement learning is then performed on each of the scenarios as follows: During the training phase, the reinforcement learning agent uses the individual replications as a state space to optimize π_k with PPO. Once it chooses to exercise the option (or the option expires) the agent traverses to a new replication and restarts the process. Total training time consists of four environment pass-overs (epochs) with four mini-batches of size $\ell * n$.

To keep the evaluations empirically consistent, out of sample testing is performed in a way that preserves the temporal structure of the data - that is, a learned policy π_k is tested on x_{k+1}. The intrinsic value generated from π_k is evaluated against the option premium - which, generally speaking, is a performance measure estimate $\hat{\theta}_k$ of some true performance measure θ_k. Similarly, this estimate (or premium) is the price attributed to the unseen testing data, x_{k+1}. Despite implementing testing this way, there are still sources of experimental error within timeframes that need to be addressed (Sec 2.5).

Unfortunately, because American options have a path dependent payoff structure, standard Monte Carlo experiments are inadequate. An alternative approach is to instead calculate the discounted value at *each time step* and take the average. More specifically, for an option with strike price K, the estimated fair price (or premium) θ_k can be estimated as follows:

$$\hat{\theta}_k^{(i)} = \frac{1}{\ell} \sum_{t=0}^{\ell} e^{-\frac{rt}{T}} max(S_{k,t}^{(i)} - K, 0)$$

Doing this across n replications yields a set of value estimates for which we can compute the expected value,

$$E[\hat{\theta}_k^{(i)}] = \frac{1}{n} \sum_{i=0}^{n} \hat{\theta}_k^{(i)}$$

[8] (Time value) = (Intrinsic value) - (Option premium)

which is the estimated performance measure $\hat{\theta}_k$. Performance measures are evaluated this way to make simulation error calculations easy. In reality, premiums might be determined with more sophisticated methods such as the Longs -taff-Schwartz[9] method.

The true value θ_k is calculated similarly, but with historical values x_k.

$$\theta_k = \frac{1}{\ell} \sum_{t=\ell k}^{\ell k+\ell} e^{-\frac{rt}{\ell}} max(X_t - K, 0)$$

This out of sample calculation can be thought of as the 'average intrinsic value' of the option x_k. A more operational interpretation of θ_k is the following: a large group of naive investors that exercise the option at random, and distribute the cumulative profit evenly. Under the principle of an *efficient market*, this is the expected payoff of all option holders. In this regard, θ_k is actually the intrinsic value of a naive policy that is based on this interpretation. This turns out to be a good policy benchmark for evaluating the *generalization error* in the final analysis.

2.5 Experimental Error

Experimental error in this analysis is categorized into three streams:

- Simulation error
- Generalization error
- Error across scenarios

Simulation error is the error measured between some real system and its simulated counterpart. More specifically, the *relative error* of simulation experiment k is given by

$$\mid \hat{\theta_k} - \theta_k \mid \leq \ \delta \mid \theta_k \mid$$

where δ is some 'acceptable' level of relative error between 0 and 1.

In the context of option pricing, the generated sample paths should closely approximate the historic realization, x_k. Unfortunately, limiting the number of usable scenarios will create a smaller sample of algorithmic performances to analyze. At the same time, the inclusion of bad scenarios might distort these results with poorly constructed learning environments. This is a very natural *exploration versus exploitation* trade-off that arises with this particular experimental design.

As it may be the case that certain historic intervals are highly noisy (e.g, Page 8), policies are developed and evaluated over all existing scenarios. The out of sample performance of these policies is contrasted with the in sample δ to analyze correlations in methodological error (Sec 3.2).

Generalization error is a measure of how accurately an algorithm can interpret and predict unseen data. In the case of option pricing, this error manifests itself in the performance of a learned exercise policy in comparison to some naive benchmark. This comparison is formalized in the way of 'excess market return' α, where exercise policies are measured for profitability. More specifically, given the cumulative out of sample reward for policy π_k, *market performance* can be characterized by

$$\alpha_k = \frac{E_{\pi_k}(\mathcal{G}(x_{k+1}))}{\theta_{k+1}}$$

where an $\alpha = 1$ implies the policy π_k matches the benchmark exactly. The generalization error is measured by a heuristic that averages these market performances with respect to the fair price $\hat{\theta}_k$. A subtle, but nontrivial assumption for this approach is that, on average, the market tends to outperform the estimated premiums (the denominator remains strictly positive):

$$\bar{\alpha} = \frac{1}{\kappa} * \frac{\sum_{k=1}^{\kappa} E_{\pi_k}(\mathcal{G}(x_{k+1})) - \hat{\theta}_k}{\sum_{k=1}^{\kappa} \theta_{k+1} - \hat{\theta}_k}$$

To mitigate sources of generalization error (and improve model accuracy) both simulated and historic data is *normalized* using Z-score standardization (i.e, each data point is reconstructed to have mean 0 and variance 1). Without normalization, the variance between gradient estimators tends to be much larger, leading to erratic policy updates. As a consequence, it is often a requirement for convergence that the state space $\mathcal{S}$ is normalized[10] under some capacity.

[9] Longstaff, A Francis. Schwartz, S Eduardo. (2001) *Valuing American Options by Simulation:A Simple Least-Squares Approach.* The Review of Financial Studies Spring 2001 Vol 14.

[10] McAllester et al. (2000) *Policy Gradient Methods for Reinforcement Learning with Function Approximation* AT&T Labs.

Error across experiments refers to the discontinuity of performance across scenarios. These errors need to be small so that the evaluation structure across scenarios is empirically consistent.

To control the error across scenarios, a single random number stream u_1[11] is selected for U in each $\hat{\theta}_k$ for $1 \leq k \leq \kappa$. More specifically, the Wiener process W_t is a transformation of i.i.d $u_1(0,1)$ random variables for each experiment belonging to Θ. This leads to a reduction in variance between scenarios by introducing positive correlation. In the case of two scenarios k_1 and k_2, the variance of the difference in performance measures $\hat{\theta}$ for an arbitrary replication j is given by:

$$\begin{aligned}\text{Var}\,(\hat{\theta}^{(j)}_{k_1} - \hat{\theta}^{(j)}_{k_2}) = \text{Var}\,(\hat{\theta}^{(j)}_{k_1}) + \text{Var}\,(\hat{\theta}^{(j)}_{k_2}) \\ -2\text{Cov}\,(\hat{\theta}^{(j)}_{k_1}, \hat{\theta}^{(j)}_{k_2})\end{aligned}$$

The use of common random numbers creates positive correlation between replications across different scenarios. As a consequence, the variance between two respective performance measures decreases.

3 Empirical Results

The empirical study is conducted using daily AAPL stock closing prices from the years 1980 to 2018. Monthly, quarterly and annual at-the-money call options are analyzed. A yearly risk free rate of 2.5% is de-annualized to the option period and used for value discounting. Individual scenarios are designed with 1000 replications and 10 observable variables[12]. These parameters are then scaled for a secondary set of experiments to inspect policy convergence with improved computational resources.

3.1 Main Empirical Findings

Several key results are highlighted from these experiments:

1. Effective exercise policies can be learned using this approach. At a panel level, simulation applied to reinforcement learning shows great promise in learning profitable exercise policies. While the empirical performance is poor with the initial setup, scaling observation spaces, replications and mini-batch size quickly improves exercise behaviours. With the most optimal design configuration, the monthly and quarterly contract sizes enjoy an out of sample $\bar{\alpha}$ greater than 1. The immediate implication is that this approach can be used effectively in options trading and other avenues of financial engineering.

2. Shorter option terms lead to better policy convergence. Even with the initial configuration, the monthly options show some convergence behaviour with an average $\bar{\alpha}$ of 0.99. As the size of the environment increases, and more observable variables are added, the average out of sample performance improves spectacularly to an α of 1.27. Similar improvements are seen for quarterly and annual contracts, although these performances are stark in comparison. Annual options, particularly, have tremendous difficulty in learning an optimal stopping policy, although it seems apparent that more replications are required for this to happen.

3. Unsuccessful policies are linked to simulation experiments with large relative errors. Comparing individual policy results with the respective simulation errors reveals a correlation between simulation error and generalization error. It may be the case that certain periods in the historical data-set are more prone to these errors due to a low signal-to-noise ratio. However, it also seems likely that training an agent with a poorly constructed environment will not allow it to appropriately generalize. With the exception of the quarterly contracts, $\bar{\alpha}$ tends to increase with the removal of high error scenarios.

Out of sample $\bar{\alpha}$		
Config.	All scenarios	$\delta \leq \delta_{50}$
$n = 1000$	0.99	+0.02
$obs = 10$	0.52	-0.10
	0.26	+.20
$n = 1000$	**1.19**	+0.02
$obs = \ell$	0.91	-0.05
	0.58	+0.06
$n = 2000$	**1.27**	+0.01
$obs = \ell$	**1.15**	+0.00
	0.77	+0.01

Table 1: *Monthly, quarterly and annual performances* ($\ell = 21, 63, 252$)

[11] Random number streams are accessed through SimRNG, a python-converted VBA codebase for generating random numbers

[12] In an MDP , the observable variables define the partially observed state space the agent is capable of interacting with

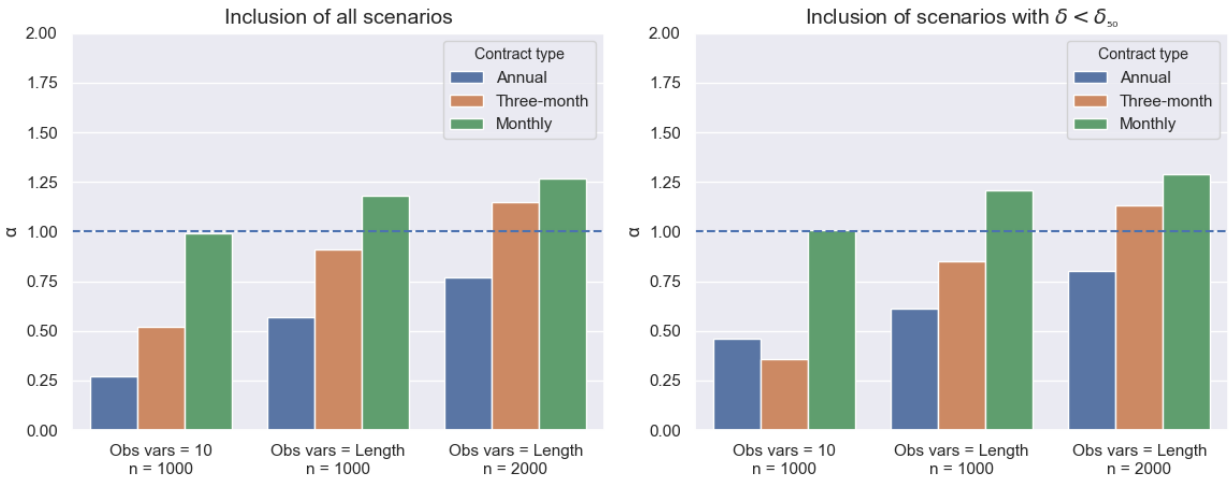


Figure 1: *Out-of-sample performances of learned policies with respect to the efficient market benchmark. Inclusion of all scenarios is shown (LHS) as well as scenarios with relative error δ less than the .5 quantile of relative errors, δ_{50} (RHS).*

3.2 Error Analysis

When $\bar{\alpha}$ is computed strictly with scenarios containing low simulation error ($\delta \leq \delta_{50}$) the generalization performance increases for the monthly and annual contracts (Table 1). This increase is more significant with the removal of bad annual scenarios. As more replications are added, the relative errors of outlying 'bad' experiments are improved, and the observed relationship becomes less significant. This suggests that a correlation between simulation error and generalization exists on some level.

Within the sample of annual contracts, two periods of historical data (2006 and 2007) elicit both high simulation errors and poor out-of-sample policy performance. When these scenarios are removed in the cutoff, the estimator $\bar{\alpha}$ improves expectantly. Within the (larger) sample of monthly contracts, the estimation of correlation is less streamlined. The performances improve in a similar fashion which suggests these bad scenarios exist , yet aren't as glaringly obvious. As a caveat, this relationship in error is nonexistent for the quarterly contracts.

3.3 Exercise behaviours

The 'exercise behaviour' of a particular policy refers to the *optimal stopping solution*, or, more generally, the observed out-of-sample *action timing*. These behaviours are illustrated for all contract lengths (Fig 2).

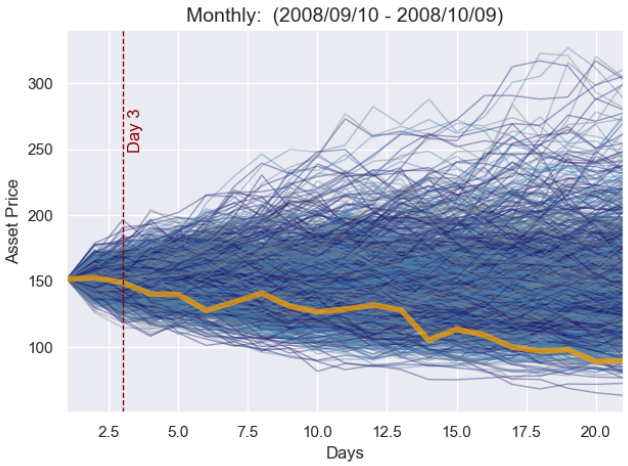
The policies trained on monthly contracts most frequently do not exercise the option at all. In the cases where the option is exercised, it is within the first 3-5 days. On the contrary, quarterly contracts most frequently exercise within the first 10-15 days, and less frequently let the contract expire. These observations are less apparent for annual contracts, although exercise behaviours tend to peak within the first 50-100 days. As a general observation, there is very little exercise behaviour within the third quarter of all contract periods.

Figure 2: *Exercise behaviours (time of exercise t with respect to option maturity T) for monthly, quarterly and annual options.*

High Performance Examples

(Units are normalized)

Blue: Monte Carlo experiment, $\hat{x}_k(\ell, n, U)$
Orange: Out-of-sample asset path, x_{k+1}
Red: Action timing for policy, π_k



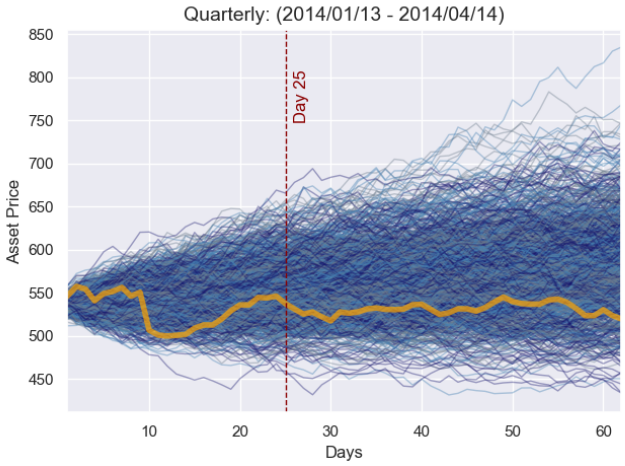
(Recession) September 10 - October 9, 2008

$\hat{\theta_k} = 0.045$

$\theta_{k+1} = 0.027$

$E_{\pi_k}(\mathcal{G}(x_{k+1})) = 0.57$

$\alpha_k = 18.60$



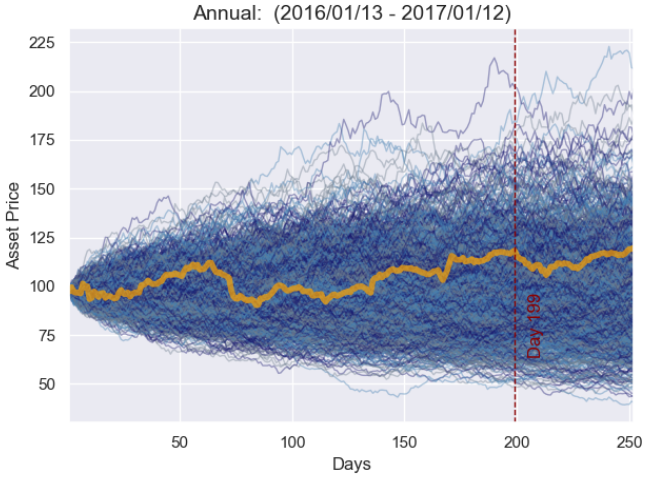
January 13 - April 14, 2014

$\hat{\theta_k} = 2.35$

$\theta_{k+1} = 1.57$

$E_{\pi_k}(\mathcal{G}(x_{k+1})) = 5.15$

$\alpha_k = 3.28$



January 13, 2016 - January 12, 2017

$\hat{\theta_k} = 3.04$

$\theta_{k+1} = 4.67$

$E_{\pi_k}(\mathcal{G}(x_{k+1})) = 11.83$

$\alpha_k = 2.53$

4 Conclusion

Developing exercise policies for American call options is an optimal stopping problem concerned with maximizing the expected option payoff. The canonical approach to said problem usually involves some flavour of Dynamic Programming. Reinforcement learning offers an alternative solution with the definition of a feasible MDP and the exploitation of large data repositories. In the case of financial data, these repositories can be scarce, which motivates the added element of stochastic simulation. In the presence of scalable computer software, this solution method becomes very practical.

The empirical results indicate that lucrative reinforcement learning policies can be learned for short term options. Out-of-sample performances are compared with an efficient market benchmark to inspect policy deployability under real market conditions. Policies learned on monthly and quarterly contracts are observed to be economically viable under this comparison. This result, however, is only achieved through computationally expensive design parameters (the reinforcement learning parameters more so than the simulation parameters).

Many of the chosen methods in this analysis are easily extendable. The random walk model that is used (vanilla GBM) can be augmented by incorporating jumps, correlated random variables and dynamic volatility. The inclusion of more observable variables, factor models and different learning architectures are other possible ways to yield better results. Furthermore, the valuation procedure can be improved to one more sophisticated, such as the infamous Longstaff-Schwartz method. The wide scope of mathematics used for this problem rapidly proliferate the possible design configurations one could use in their approach.

Even so, the simplistic design showcased in this analysis shows promising results - which may be an indication that these methods are well poised for financial engineering. Machine learning methods of analysis should be cautiously used in financing engineering. Data quality and data availability are well established concerns that encumber these methods greatly. Synthetic data is one possible solution that offers a trade-off between quality and quantity in learning-based approaches.

References

Chen, Luyang Pelger, Markus Zhu, Jason. (2019). *Deep Learning in Asset Pricing.* SSRN Electronic Journal. 10.2139/ssrn.3350138.

Dhariwal, Prafulla. Klimov, Oleg. Radford, Alec. Shulman, John. Wolski, Filip. (2017) *Proximal Policy Optimization Algorithms* OpenAI, 1707.06347.

Dmouj, Abdelmoula. (2006) *Stock Price Modelling: Theory and Practice* Vrije University, Amsterdam.

Gu, Shihao. Kelly, Bryan. Xiu, Dacheng. (2019) *Empirical Asset Pricing via Machine Learning*, University of Chicago

Hull, John C. (2008). *Options, Futures and Other Derivatives* 7th edition.

Li, Yuxi. Szepesvari, Csaba. Schuurmans, Dale. (2009) *Learning Exercise Policies for American Options.* University of Alberta.

Longstaff, A Francis. Schwartz, S Eduardo. (2001) *Valuing American Options by Simulation:A Simple Least-Squares Approach.* The Review of Financial Studies Spring 2001 Vol 14.

Mansour, Yishay. McAllester, David. Singh, Satinder. Sutton S, Richard. (2000) *Policy Gradient Methods for Reinforcement Learning with Function Approximation* AT&T Labs Research.

Ng Y, Andrew. (2003) *Shaping and policy search in Reinforcement Learning* University of California, Berkeley.

Omer B, Sezer. Murat, Ozbayoglu. M. Ugur, Gudelek. (2019) *Financial Time Series Forecasting with Deep Learning : A Systematic Literature Review* University of Economics and Technology, Ankara, Turkey 1911.13288v1.

Code Appendix

There are three Python files:

`main.py` is the primary codebase that calls on the other two Python files to run experiments. After reading in the data, simulation and reinforcement learning is performed for each scenario and the out-of-sample testing is recorded as a csv file.

`simEnv.py` reads, cleans, and extracts the data for each scenario as an object. During the simulation step, simulation paths are constructed as an $nx\ell$ matrix with the added observables variables. This matrix is used in the reinforcement learning step, as well as in calculating the option price.

`optionEnv.py` is the reinforcement learning environment. This environment defines the MDP and is passed as a vectorized environment in `main.py` for the training phase. Then, the trained model uses the same environment for the out-of-sample testing on the historical data.

The open source libraries `OpenAI` and `Stable Baselines` are used for reinforcement learning with `tensorflow` as a backend.

main.py

```
from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import PPO2
from env.simEnv import Equity
from env.SimRNG import Normal
from env.optionEnv import OptionsEnv
import pandas as pd
import tensorflow as tf
import os
import warnings
warnings.filterwarnings('ignore')
tf.logging.set_verbosity(tf.logging.ERROR)
os.environ['KMP_WARNINGS'] = 'off'

os.chdir('C:/Users/jeree/PycharmProjects/AmericanOptionsEnv/data')
AAPL = pd.read_csv('AAPL.csv')
AAPL.name = 'Apple'

# Experiment design parameters
ir = .025
numSamplePaths = 1000
optionLength = [21, 63, 252]
theta_k = 1

for o in optionLength:

    numSamplePaths = numSamplePaths * 2

    U = []
    for i in range(0, numSamplePaths * o):
        U.append(Normal(0, 1, 1))

    numObservables = o
    profits, day_exercised, fair_prices, naive_prices, dates = ([] for _ in range(5))

    # Create object using AAPL data
    obj = Equity(AAPL, o, numSamplePaths, numObservables, ir)

    for d in (obj.get_params()['Start Date'][:-1]):

        # Simulate data for a given date and option length
        sim_df = obj.simulate(date=d, rvs=U)
        real = obj.get_real(date=d)[numObservables:]

        # Get fair option price with the simulated data
        fair_price, naive_price = obj.get_prices(sim_df)
        fair_prices.append(fair_price)
        naive_prices.append(naive_price)

        estimated_theta_k = fair_price
        rel_error = abs(theta_k - estimated_theta_k) / abs(theta_k)
        theta_k = naive_price

        # The algorithm requires a vectorized environment to run
        train_env = DummyVecEnv([lambda: OptionsEnv(sim_df, numObservables, ir, train=True)])

        print('\nDate: ', d)
        print('Initial Price: ', real[0])
        dates.append(d)

        # Model is trained via Proximal Policy Optimization
        model = PPO2(MlpPolicy, train_env, verbose=0)
        model.learn(total_timesteps=numSamplePaths*o)
        train_env.close()

        # Out of sample testing
        test_env = DummyVecEnv([lambda: OptionsEnv(sim_df, numObservables, ir, train=False)])
        obs = test_env.reset()
        exercised = False
        day = 1
        while not exercised:
            action, states = model.predict(obs)
            obs, reward, done, info = test_env.step(action)
            if done:
                exercised = True
                print('Day ', day)
                day_exercised.append(day)
                print('Profit: ', float(reward))
                print('Fair price: ', fair_price)
```

main.py

```
                print('Naive price: ', naive_price)
                print('Rel error: ', rel_error)
                profits.append(float(reward))
            day += 1
        test_env.close()

    data = {'Dates': dates,
            'Fair Prices': fair_prices,
            'Naive Premium': naive_prices,
            'RL Premium': profits,
            'Day exercised': day_exercised}
    df = pd.DataFrame(data)
    df.name = 'MoreSims SO Option Length ' + str(o)
    df.to_csv(df.name, sep='\t')
```

simEnv.py

```python
import pandas as pd
import numpy as np
import math


class Equity:

    def __init__(self, df, option_length, num_sims, num_observables, ir):

        # Initialization
        self.name = df.name
        self.numSamplePaths = num_sims
        self.numObservables = num_observables
        self.interest_rate = ir
        self.option_length = option_length
        self.df = df[['Date', 'Close']]
        self.sim_mean = 0
        self.sim_std = 0

    def get_params(self):

        params_df = pd.DataFrame(columns=['Start Date', 'Initial Price', 'Mu', 'Sigma', 'Path'])
        disposable = self.df

        # Creation of option scenarios with sliding window = option length
        for i in range(0, int(np.floor(self.df.shape[0] / self.option_length))):

            period = disposable.head(n=self.option_length*2)
            disposable = disposable.iloc[self.option_length:]

            prev = period.iloc[:self.option_length]
            actual = period.iloc[self.option_length:]

            real_path = np.append(
                prev['Close'].to_numpy()[-self.numObservables:], actual['Close'].to_numpy())
            start_date = actual['Date'].to_numpy()[0]
            initial_price = actual['Close'].to_numpy()[0]
            returns = prev['Close'].pct_change()
            mu = np.mean(np.log1p(returns))
            sigma = np.std(np.log1p(returns), ddof=1)

            params_df.loc[i] = [start_date] + [initial_price] + [mu] + [sigma] + [real_path]

        return params_df

    def get_real(self, date):

        params = self.get_params()

        # Get real asset path with n observables
        if date not in params['Start Date'].to_numpy():
            print('Invalid date')
            return
        else:
            idx = params['Start Date'][params['Start Date'] == date].index[0]
            test_data = params['Path'][idx]
            return list(test_data)

    def simulate(self, date, rvs):

        params = self.get_params()

        # Simulate sample paths for given date
        if date not in params['Start Date'].to_numpy():
            print('Invalid date')
            return
        else:
            idx = params['Start Date'][params['Start Date'] == date].index[0]
            observables = params['Path'][idx][:self.numObservables]
            drift = params['Mu'][idx]
            volatility = params['Sigma'][idx]
            df = pd.DataFrame()
            cols = []

            # Construction of dataframe containing all sample paths (replications)
            j = 0
            for n in range(self.numSamplePaths):
                X = params['Initial Price'][idx]
                value_list = [X]
```

simEnv.py

```
 80
 81                # Discretized Geometric Brownian Motion
 82                for s in range(self.option_length-1):
 83                    Z = rvs[j]
 84                    X = float(X * np.exp(drift + volatility * Z))
 85                    value_list.append(X)
 86                    j += 1
 87                df[n] = list(observables) + list(value_list)
 88                cols.append('Sim ' + str(n + 1))
 89
 90            # Attach real data at the end
 91            df['Real'] = self.get_real(date)
 92            cols.append('Real')
 93
 94            # Normalization of the form (x - mu) / sigma
 95            self.sim_mean = pd.Series.mean(df.mean(axis=None))
 96            self.sim_std = pd.Series.std(df.std(axis=None))
 97            df = pd.DataFrame(
 98                np.divide(np.subtract(df.to_numpy(), self.sim_mean), self.sim_std))
 99
100            df.columns = cols
101
102            return df
103
104    def get_prices(self, df):
105
106        real = df['Real'].iloc[self.numObservables:].to_numpy()
107        df = df.iloc[self.numObservables:].drop('Real', axis=1).reset_index(drop=True)
108        strike_price = df['Sim 1'].iloc[0]
109        fair_values = []
110        naive_values = []
111
112        # Compute fair price with in-sample simulation
113        # Compute naive price with out-of-sample real path
114        t = 1
115        for index, row in df.iterrows():
116            arr = np.multiply(np.maximum(
117                    row.to_numpy() - strike_price, np.zeros(
118                        self.numSamplePaths)), pow(math.exp(-self.interest_rate / len(df.index)), t))
119            fair_values.append(np.mean(arr))
120            naive_values.append(
121                max(real[t-1] - strike_price, 0) * pow(
122                    math.exp(-self.interest_rate / len(df.index)), t))
123            t += 1
124        fair_price = np.mean(fair_values)
125        naive_price = np.mean(naive_values)
126
127        return fair_price, naive_price
128
```

optionEnv.py

```python
import math
import gym
from abc import ABC
from gym import spaces
import pandas as pd
import numpy as np
from env.SimRNG import Uniform


class OptionsEnv(gym.Env, ABC):
    metadata = {'render.modes': ['human']}

    def __init__(self, df, numObservations, interest_rate, train):
        super(OptionsEnv, self).__init__()

        # Initialize environment
        self.train = train
        self.df = df
        self.numObservations = numObservations
        self.current_sim = 'Sim 1'
        self.current_step = 0
        self.strike_price = self.df[self.current_sim].iloc[self.numObservations]
        self.gamma = math.exp(-interest_rate / len(self.df.index))
        self.reward_range = (0, pd.Series.max(self.df.max(axis=None)))

        # Actions of the format: exercise, don't exercise
        self.action_space = spaces.Box(
            low=np.array([0]), high=np.array([2]), dtype=np.float16)

        # Price values of the last n observations
        self.observation_space = spaces.Box(
            low=0, high=1, shape=(1, self.numObservations+1), dtype=np.float16)

    def next_observation(self):

        # Get the price data from the last n periods and scale
        obs = np.array([
            self.df.loc[
                self.current_step: self.current_step + self.numObservations, self.current_sim].values])
        return obs

    def step(self, action):

        current_price = self.df.loc[self.current_step + self.numObservations, self.current_sim]
        reward = 0
        done = False

        if action <= 1:  # CHOOSE NOT TO EXERCISE
            reward = 0
            done = False

        elif action <= 2:  # CHOOSE TO EXERCISE
            reward = max(current_price - self.strike_price, 0) * pow(self.gamma, self.current_step)
            done = True

        self.current_step += 1

        if self.current_step + self.numObservations == len(self.df.index):
            reward = max(current_price - self.strike_price, 0) * pow(self.gamma, self.current_step)
            done = True

        obs = self.next_observation()

        return obs, reward, done, {}

    def reset(self):

        if self.train:
            randsim = int(Uniform(1, len(self.df.columns) - 1, 1))
            self.current_sim = 'Sim '+str(randsim)
        else:
            self.current_sim = 'Real'
        self.current_step = 0

        return self.next_observation()



```