# *Implementation of an Actor Critic Algorithm for Mortgage Refinancing*

*Kaivalya Rawal (2015A7PS0005G)  Samarth Mehrotra(2015A7PS0062G)*

***Abstract*—The aim of the project was to implement an actor critic algorithm and apply it to the field of Optimal Mortgage Refinancing. The paper that was implemented is: A Simultaneous Deterministic Perturbation Actor-Critic Algorithm with an Application to Optimal Mortgage Refinancing (2006).**

***Keywords—Actor-critic, reinforcement learning, mortgage refinancing***

## I. Introduction

There are many sequential decision tasks in which the consequences of an action emerge at a multitude of times after the action is taken and the problem is to find good strategies for selecting actions based on both their short and long term consequences. Such tasks are encountered in many fields such as economics, manufacturing, and artificial intelligence. These are usually formulated in terms of a dynamical system whose behavior unfolds over time under the influence of a decision maker's actions. The randomness involved in the consequences of the decision maker's actions is taken care of by modeling the dynamical system as a controlled stochastic process. Markov Decision Processes (MDP) are a natural choice to model such systems and Dynamic Programming (DP) is a general methodology for solving these. DP, however, requires complete knowledge of transition probabilities. Moreover, the computational requirements using DP are high in the presence of large state space. Recently there has been a lot of interest in simulation-based Reinforcement Learning (RL) algorithms for solving MDPs. These algorithms neither use transition probabilities nor estimate them and are useful in general for finding optimal control strategies in real life systems for which model information is not known. There are a certain class of (RL-based) algorithms that go under the name of actor-critic algorithms. These can be viewed as stochastic approximation versions of the classical policy iteration technique for solving MDPs.

### *A. Objective*

The aim of our project was to write a python based implementation of the proposed algorithm, and then test it using a simulated environment as described in the paper introducing the algorithm.

### *B. Theoretical Background*

Reinforcement learning (RL) is an area of machine learning inspired by behaviourist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic

algorithms. In the operations research and control literature, reinforcement learning is called approximate dynamic programming. The approach has been studied in the theory of optimal control, though most studies are concerned with the existence of optimal solutions and their characterization, and not with learning or approximation In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

In machine learning, the environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible.

The description of a Markov Decision Process in the paper as follows:

Consider a process, observed at time epochs t = 0, 1, ...,

to be in one of the states i ∈ S. Let S = {1, 2, . . . , s } denote

the state space. After observing the state of the process, an

action a ∈ A = {a 0 , a 1 , . . . , a |A| } is taken, where A is the

set of all possible actions. If the process is in state i at time

n and action a is chosen, then two things happen: (1) we

receive a finite reward R(i, a) and (2) the next state of the

system is chosen according to the transition probabilities

$P_{ij}$ (a). We let $X_n$ denote the state of the process at time

n and a n the action chosen at that time.

We assume that | R(i, a) |< M ∀ i,a. An admissible

policy or simply a policy is any rule for selecting feasible

actions. An important subclass of policies is the class

of stationary policies. A policy is said to be stationary

if the action it chooses at any time n depends only on

the state of the process at that time. Hence, a stationary

policy is a function π : S → A. We assume for ease

of exposition that all actions are feasible in each state. A

stationary randomized policy can be considered as a map

φ : S → P(A) (P (...) = the space of probability vectors on

"..."), which gives the conditional probabilities of $a_j$ given $X_n$ for all $0 \leq j \leq |A|$. The objective in reinforcement learning is to find an optimal value for this, so as to maximise reward gained by the agent in the long run.

## II. Python Libraries and Programming Concepts Used

### *A. NumPy*

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

The core functionality of NumPy is its "ndarray", for n-dimensional array, data structure. These arrays are strided views on memory. In contrast to Python's built-in list data structure (which, despite the name, is a dynamic array), these arrays are homogeneously typed: all elements of a single array must be of the same type.

Such arrays can also be views into memory buffers allocated by C/C++, Cython, and Fortran extensions to the CPython interpreter without the need to copy data around, giving a degree of compatibility with existing numerical

libraries. This functionality is exploited by the SciPy package, which wraps a number of such libraries (notably BLAS and LAPACK). NumPy has built-in support for memory-mapped ndarrays.

A few examples:

```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> x
array([1, 2, 3])
>>> y = np.arange(10)  # like Python's range, but returns an array
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a = np.array([1, 2, 3, 6])
>>> b = np.linspace(0, 2, 4)  # create an array with four equally spaced points starting with 0 and ending with 2.
>>> c = a - b
>>> c
array([ 1.        , 1.33333333, 1.66666667, 4.        ])
>>> a**2
array([ 1,  4,  9, 36])
```

NumPy arrays were chose for their speedy execution and ease of use to represent the Hadamard matrix and the policy vectors for each state.

*B. Object Oriented Programming*

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

Languages that support object-oriented programming typically use inheritance for code reuse and extensibility in the form of either classes or prototypes. Those that use classes support two main concepts:

Classes – the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods) themselves, i.e. classes contain the data members and member functions

Objects – instances of classes

Objects sometimes correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product". Sometimes objects represent more abstract entities, like an object that represents an open file, or an object that provides the service of translating measurements from U.S. customary to metric.

Each object is said to be an instance of a particular class (for example, an object with its name field set to "Mary" might be an instance of class Employee). Procedures in object-oriented programming are known as methods; variables are also known as fields, members, attributes, or properties. This leads to the following terms:

Class variables – belong to the class as a whole; there is only one copy of each one

Instance variables or attributes – data that belongs to individual objects; every object has its own copy of each one

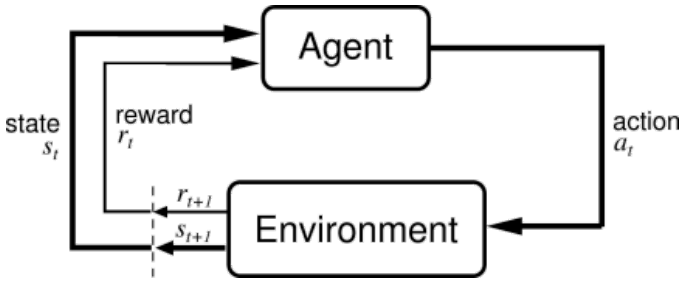
Member variables – refers to both the class and instance variables that are defined by a particular class

Class methods – belong to the class as a whole and have access only to class variables and inputs from the procedure call

Instance methods – belong to individual objects, and have access to instance variables for the specific object they are called on, inputs, and class variables

Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively pointers, serving as actual references to a single instance of said object in memory within a heap or stack. They provide a layer of abstraction which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a constructor. A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data.

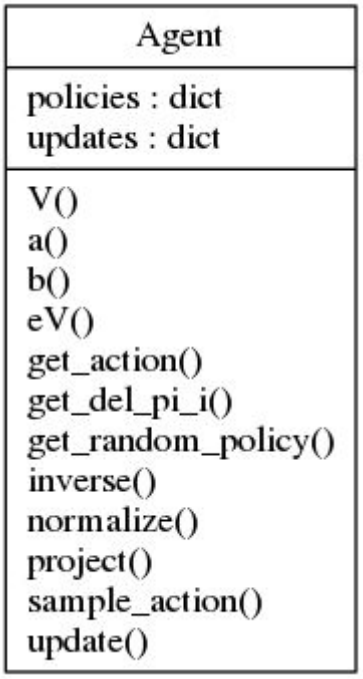
## III. METHODOLOGY




The software designed was specifically made so as to maintain the Agent - Environment separation as in this figure and all reinforcement learning algorithms. We did this by writing entirely separate modules for the Agent and the Environment, in different python files that don't have any interaction with each other at all.

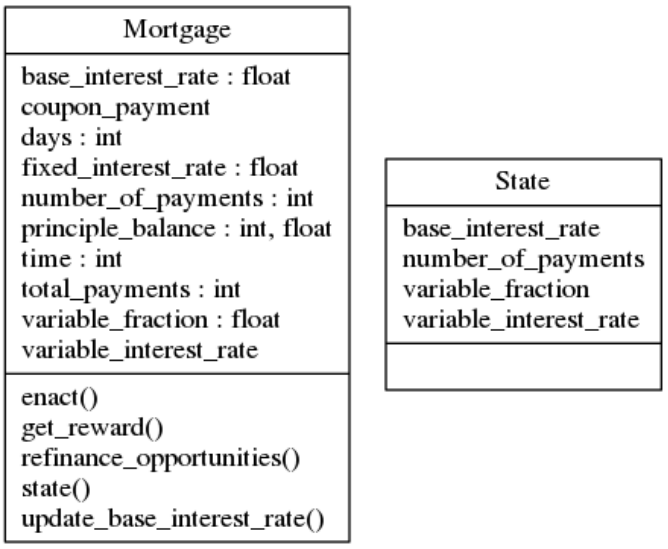
A Python script was written to implement the algorithm and test it according to the specified environment in **simulator.py**. The module for the agent was written in **agent.py**, and the environment in **environment.py**.

agent.py represents the mortgagor, i.e. the borrower. mortage.py represents the current mortgage contract and the environment. simulator.py serves as a driver script to perform the final simulation.




**UML description of the Agent module**

agent.py maintains a python dictionary where the keys are 'states' and values are the associated probability vector for each state. When the get_action method is called, the agent returns the action as per the probability vector.




**UML description of the Mortgage module**

mortgage.py maintains all the details about the current mortgage contract, i.e. principal balance, variable and fixed interest rates etc. When the enact function of a mortgage is called, it is either refinanced or not-refinanced as per the action suggested by the agent.

Please refer to the code attached at the end of this report for more details.

We made certain assumptions while implementing the algorithm. These were:

1. Equation 7 of the paper refers to an arbitrary integer 'L'. We define L (=1) at the start of the algorithm and set it to a constant value which is not updated again.
2. The calculation of the value function involves a lookahead of the expected reward. We make a one-step look ahead in our approach.
3. We define delta to be equal to 0.01 (Equation 7)
4. We set alpha to 0.05 (Equation 8)
5. We assume that in the first mortgage, the variable fraction is zero.
6. In equation 7, we store a separate count variable 'n' for each state rather than a global count variable. This is to ensure faster convergence.
7. The series that we have assumed for both a(n) and b(n) is 1/(n+1).
8. According to us if the current variable fraction is equal to the proposed variable action, i.e. the suggested action, then the mortgage is not refinanced.
9. All the lenders in the market offer the same market rates of interest (rt and vt).

## IV. Conclusion and Future Work

Future work can include implementing other actor critic algorithms using a similar strategy. It can also include verifying the correctness of the algorithm on real world data from Yahoo Finance. One can also try using more recent reinforcement learning algorithms in the same scenario of mortgage refinancing, and compare their performance to the current algorithm.

## Acknowledgment

We would like to thank Dr. Mayank Goel for giving us this opportunity to work on this project.

## References

[1] Stuart J. Russell and Peter Norvig. 2003. Artificial Intelligence: A Modern Approach (2 ed.). Pearson Education.

[2] Puterman. Markov Decision Processes. Wiley Inter-science, 1994.

[3] S.M. Ross. Introduction to Stochastic Dynamic Programming. Academic Press, 1983.

[4] V. R. Konda and V. S. Borkar. Actor-critic type learning algorithms for Markov decision processes. SIAM Journal on Control and Optimization, 38:94–123, 1999.

[5] S. Bhatnagar, M. C. Fu, S. I. Marcus, and I. J. Wang. Two-timescale simultaneous perturbation stochastic approximation using deterministic perturbation sequences. ACM Transactions on Modeling and Computer Simulation, 13(2):180–209, 2003.

## agent.py

```
from mortgage import State
from scipy.linalg import hadamard
import math
import random
import numpy as np



delta = 0.01
L = 1
alpha = 0.05

C = 2**(math.ceil(math.log(12, 2))) # C = 16
hadamard = hadamard(C)
hbar = hadamard[:, 1:12]
offset = random.randint(0,15)

class Agent():

    def __init__(self):
        self.policies = {} # dictionary mapping states to policies (s ->
           pi_i)
        self.updates = {}

    def sample_action(self, policy):
        return np.random.choice([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
           0.8, 0.9, 1.0], p=policy)

    def get_random_policy(self):
        policy = np.random.random_sample((11,))
        policy = Agent.normalize(policy)
        print(policy)
        #print(sum(policy))
        return policy

    def normalize(v):
        norm = np.linalg.norm(v, ord=1)
        if norm == 0:
           return v
        return v / norm

    def get_action(self, opportunities, state):
        policy = self.policies.get(state)
        if policy is None:
            policy = self.get_random_policy()
            self.updates[state] = 0
            self.policies[state] = policy
        if not opportunities:
            print("No refinance because no opportunity, returning ", state
               .variable_fraction)
            return state.variable_fraction # same fb implies no
               refinancing
```

```
    #print("updatedict = ", self.updates)
    result = self.sample_action(policy)
    #Agent.update_policy(state, result)
    print("Sampled action with given policy is: ", result)
    return result

def update(self, action, reward, state, time_passed,
   current_coupon_payment):
    print("updating state = {}".format(state))
    n = self.updates[state]
    policy = self.policies[state]
    print("previous policy = ", policy)
    print("n (iteration value) = ", n)
    a = Agent.a(n)
    inverse_del_pi_i = Agent.inverse(Agent.get_del_pi_i(n))
    print("inverse del pi i = ", inverse_del_pi_i)
    v_n_l = Agent.V(n, state, reward, time_passed,
       current_coupon_payment)
    print("Vnl = ", v_n_l)
    new_policy = Agent.project(policy + a*(v_n_l/delta)*
       inverse_del_pi_i)
    print("new policy = ", new_policy)
    self.policies[state] = new_policy
    self.updates[state] = n+1

def get_del_pi_i(n):
    return hbar[(n+offset)%C]

def inverse(del_pi_i):
    #print(del_pi_i)
    #print(del_pi_i.shape)
    return del_pi_i

def a(n):
    a = 1/(n+1)
    return a

def b(n):
    b = 1/(n+1)
    return b

def V(n, state, reward, time_passed, current_coupon_payment):
    if n == 0: # base case
        return 2
    b = Agent.b(n)


    del_pi_i = Agent.get_del_pi_i(n)
    pi_bar = Agent.project(self.policies[state]+delta*del_pi_i)
    new_fb = self.sample_action(pi_bar)

    c = current_coupon_payment
    fb = state.variable_fraction
    vt = state.variable_interest_rate
```

```
        rt = state.base_interest_rate # should be mt!
        days = 30

        if fb == new_fb: # no expected refinance
            e_reward = c + c*fb*vt + c*(1-fb)*rt
        else:
            e_reward = c + c*fb*vt*(time_passed/days) + c*(1-fb)*rt*(
               time_passed/days)
        val = Agent.eV(n-1, None, e_reward)


        return (1-b)*Agent.V(n-1, state, reward, time_passed,
           current_coupon_payment) + b*(reward + alpha*( val ))

    def eV(n, state, reward):
        if n == 0: # base case
            return 2
        b = Agent.b(n)
        return (1-b)*Agent.eV(n-1, state, reward) + b*(reward)

    def project(policy):
        minimum = np.amin(policy)
        if (minimum < 0):
            policy += abs(minimum)
        projection = Agent.normalize(policy)
        return projection
```

## mortgage.py

```
import numpy as np
import random

class State():

    def __init__(self, base_interest_rate, variable_fraction,
       variable_interest_rate, number_of_payments):
        self.base_interest_rate = base_interest_rate
        self.variable_fraction = variable_fraction
        self.variable_interest_rate = variable_interest_rate
        self.number_of_payments = number_of_payments

    def __hash__(self):
        return hash((self.base_interest_rate, self.variable_fraction, self
           .variable_interest_rate, self.number_of_payments))

    def __str__(self):
        return 'rt:{} fb:{} vt:{} n:{}'.format(self.base_interest_rate,
           self.variable_fraction, self.variable_interest_rate, self.
           number_of_payments)

class Mortgage():

    def __init__(self):
        self.principle_balance = 200000
        self.total_payments = 60
        self.days = 30 # total expected loan duration is 30*60 days
        self.coupon_payment = self.principle_balance / self.total_payments
            # might require more than N payments!
        self.time = 30

        self.base_interest_rate=0.06 # rt
        b = 0.005
        self.fixed_interest_rate = self.base_interest_rate + b # mt = rt+b
        self.variable_fraction = 0.0 #fb
        c = random.choice([2.0,2.5,3.0,3.5,4.0])
        self.variable_interest_rate = self.base_interest_rate + c # vt =
           rt + c
        self.number_of_payments = 0

    def enact(self, action):
        reward = self.get_reward(action)
        self.number_of_payments += 1
        self.principle_balance -= self.coupon_payment
        print("New balance after coupon payment: ", self.principle_balance
           )
        self.principle_balance += self.principle_balance*self.
           variable_fraction*(self.variable_interest_rate/12) + self.
           principle_balance*(1-self.variable_fraction)*(self.
           fixed_interest_rate/12)
        # compounding monthly, by 1/12th of the annual interest rate
```

```
        print("New balance after interest accrues: ", self.
            principle_balance)
        if self.principle_balance == 0:
            return "Loan completely paid back."

        print("Enacting action: ", action)
        if action == self.variable_fraction: # staying in the same state
            print("Staying in same state")
            pass
        else:
            print("Updating state... ")
            self.update_base_interest_rate()
            print("Updated rt: ", self.base_interest_rate)
            print("Updated mt: ", self.fixed_interest_rate)
            self.variable_fraction = action
            print("Updated fb: ", self.variable_fraction)
            self.number_of_payments = 0
            self.principle_balance *= 1.02 # transaction cost = 2%
            print("New pb after transaction cost: ", self.
                principle_balance)
        c = random.choice([2.0,2.5,3.0,3.5,4.0])
        self.variable_interest_rate = self.base_interest_rate + c
        print("New vt:", self.variable_interest_rate)
        return reward

    def get_reward(self, action):
        c = self.coupon_payment
        fb = self.variable_fraction
        vt = self.variable_interest_rate
        rt = self.base_interest_rate # should be mt!
        pb = self.principle_balance
        if self.variable_fraction == action: # no refinance
            self.time += 30
            return c + c*fb*vt + c*(1-fb)*rt
        else:
            result = 0.02*(pb-c) + c + c*fb*vt*(self.time/self.days) + c
                *(1-fb)*rt*(self.time/self.days)
            self.time = 0
            return result


    def refinance_opportunities(self):
        result = None
        if (np.random.poisson(lam=(1.0))>0):
            result = True
        else:
            result = False
        print("Refinance opportunities in poisson distribution with k = 1:
            ", result)
        return result

    def state(self):
        result = State(self.base_interest_rate, self.variable_fraction,
            self.variable_interest_rate, self.number_of_payments)
```

```
        print(result)
        return result


    def update_base_interest_rate(self):
        a=0.005
        multiplier = [-2.0, -1.0, 0.0, 1.0, 2.0]
        selected = random.choice(multiplier)
        print('updating rt by setting = rt+({}*a)'.format(selected))
        rt = self.base_interest_rate+(a*selected)

        if rt < 0.04:
            self.base_interest_rate = 0.04
        elif rt > 0.12:
            self.base_interest_rate = 0.12
        else:
            self.base_interest_rate = rt
        b = 0.005
        self.fixed_interest_rate = self.base_interest_rate + b
```

## simulator.py

```
from mortgage import Mortgage
from agent import Agent
import numpy as np
import random
import timeit

class Simulator():

    def __init__(self):
        self.m = Mortgage()

    def simulate():
        s = Simulator()
        a = Agent()
        for i in range(10):
            print('\n\n------')
            print('\ngetting action from agent')
            state = s.m.state()
            action = a.get_action(s.m.refinance_opportunities(), state)
            print("\nenacting action in environment")
            reward = s.m.enact(action)
            print("\nupdating agent policies based on reward = ", reward)
            a.update(action, reward, state, s.m.time, s.m.coupon_payment)



def main():
    np.random.seed(0)
    random.seed(0)
    Simulator.simulate()

if __name__ == '__main__':
    main()
```