A Deep Reinforcement Learning Approach to Asset-Liability Management

Alan Fontoura
CEFET/RJ - PPCIC
Rio de Janeiro, Brazil
alan.fontoura@eic.cefet-rj.br

Diego Barreto Haddad
CEFET/RJ - PPCIC/PPEEL/PPGIO
Rio de Janeiro, Brazil
diego.haddad@cefet-rj.br

Eduardo Bezerra
CEFET/RJ - PPCIC
Rio de Janeiro, Brazil
ebezerra@cefet-rj.br

***Abstract*—Asset-Liability Management (ALM) is a technique used to optimize investment portfolios, considering a future flow of liabilities. Its stochastic nature and multi-period decision structure favors its modeling as a Markov Decision Process (MDP). Reinforcement Learning is a state-of-the-art group of algorithms for MDP solving, and with its recent performance boost provided by deep neural networks, problems with long time horizons can be handled in just a few hours. In this paper, we address an ALM problem with a variation of Deep Deterministic Policy Gradient algorithm. Opposed to all other approaches in the literature, our model does not use scenario discretization, which is a major contribution to ALM study. Our experimental results show that the Reinforcement Learning framework is well fitted to solve this kind of problem, and has the additional benefit of using continuous state spaces.**

***Index Terms*—ALM, reinforcement learning, deep deterministic policy gradient**

I. Introduction

One of the most studied issues in finance is asset management. Usually seen as a portfolio optimization problem, there are two common approaches to it: trying either (1) to minimize the portfolio's chosen risk measure given an expected return, or (2) to maximize the expected return, while not surpassing a certain risk level.

Asset-liability management (ALM), on the other way, is a more complex matter. The investor's goal is to fulfill a series of obligations, which may or may not be stochastic in nature. These obligations (or liabilities) usually are correlated to one or more of the assets available to the decision maker. In this scenario, one can not just aim for optimizing the risk-return relation: investments have also to match (or preferably, outperform) liabilities, respecting their due dates.

Most obvious uses for ALM strategies lie in the financial sector, specially in banking, insurance companies and pension funds. Nonetheless, any individual who saves money for a future debt (a person who wants to pay for his children's college, for example) may benefit from its techniques.

The structure of an ALM problem is simple: at each time step, there is an *asset*, which is the total amount to be invested, and a set of available investments. An allocation of the asset is chosen among all (or part) of the available investments, and after one time step, the incomes are added to the original amount. The *liability* for the given time step is paid, and a new allocation happens. This process is repeated until there are no more assets (hence generating a *deficit*) or liabilities (and there is a *surplus*).

The above structure can be modeled as a *Markov Decision Process* (MDP) [1]. In such a process, there is an initial state, which is observed by an agent. At each time step, this agent takes an action, and based on current state-action pair, the agent transits to a new state s' and receives a reward provided by the environment. With this new state s' observed, a new action is taken by the agent. This process is repeated, until a terminal condition happens. The goal of the agent is to maximize the expected sum of received rewards.

The *Reinforcement Learning* (RL) framework [1] is a group of machine learning algorithms to solve MDPs. Its main idea is to associate a *reward* to every state-action-next state tuple, and then find a *policy* (i.e., a function which maps states to actions) that maximizes the expected value of these rewards sum, along the whole process. In this paper, we adapt a particular Deep RL algorithm (known as *Deep Deterministic Policy Gradient*, or DDPG) [2] to solve an ALM problem. Fifteen asset/liability simulations have been run, in order to measure algorithm's performance against different levels of debts and time horizons. Results are consistent with what was expected: greater debts imply in smaller payment capacity, and longer time spans lead to higher uncertainty. This paper has two major contributions:

- our proposed model uses continuous state spaces, instead of discrete scenario trees;
- to the best of author's knowledge, this is the first advanced method to solve ALM problems using a reinforcement learning approach.

The remaining of this paper is structured as follows: Section II describes Asset-Liability Management, and its most common approach. Section III introduces the basics of reinforcement learning and DDPG concepts. Section IV describes our proposed RL model for ALM, whose performance is assessed in Section V. Section VI presents the concluding remarks.

II. Asset-Liability Management

The main goal of an Asset-Liability Management model is to find an optimal investment strategy, while considering the flow of liabilities. To further clarify ALM dynamics, let us consider a particular scenario: a closed pension fund has

2643-6264/19/$31.00 ©2019 IEEE
DOI 10.1109/BRACIS.2019.00046

gathered one billion monetary units through contributions of its participants. This amount has to be invested, and will later be used to pay their retirements. The fund administration should pay each retired participant until his/her death. To do so, they estimate how much has to be paid each year, and until when. This estimation is based on the number of living pensioners (using actuarial tables) and how much they receive per year.

For example, in next year, fund's administration expects to have 500 pensioners alive, with an average year income of 100.000 monetary units: so, this period's debt is estimated in 50 millions. In the following year, 60 more shall retire, and actuarial table indicates 10 will perish. So, estimated debt for this year will be 55 millions. These year estimates will grow up to a certain point, when the amount of living retired participants reaches its peak, and then slowly shrink to zero.

In the above scenario, a risky investment strategy could lead to losses that would let lots of pensioners without their incomes. On the other way, a too conservative strategy may not make enough money to honor fund's debts. Thus, the choice of an adequate investment strategy is of paramount importance to the fund's success and must consider total assets, available investments and liability flow **together**. In order to keep this balance, it's desired to maintain the ratio between total assets and liability's present value always close to 1. Liability's present value is given by $\sum_{i=1}^{T} \frac{L_i}{(1+r)^i}$, where L_i is the estimated debt at time period i, and r is a *discount rate*. This rate is the interest total assets are supposed to grow each time period.

It is shown by Brinson, Hood and Beebower [3] that the optimal investment strategy depends mainly on the investment policy, which determines how much should be allocated in the major investment groups (stocks, bonds or cash). Individual securities selection, market timing and costs (the other elements analysed) had a very small impact in the pension funds examined. Therefore, most ALM models try to find the optimal investment policy, given a certain set of regulatory constraints.

Due to its stochastic nature, and a need for dynamic decision making, a very popular approach to ALM is Multistage Stochastic Programming (MSP). One of the first works with such a model is the one proposed by Bradley and Crane [4], for a bond portfolio optimization. Carino et al. [5] proposed a model for a Japanese insurance company, and Carino, Myers and Ziemba [6] revisited this same model four years later. Hilli, Koivu, Pennanen and Ranne [7], Valladao and Veiga [8], and Haneveld, Streutker and Van der Vlerk [9] created models for a Finnish, a Brazilian and a Dutch pension funds, respectively. Gulpinar and Pachamanova [10] worked with multi-period robust optimization.

One thing all these models (and many more) have in common is they need to generate a scenario tree to handle uncertainty. According to Defourny, Ernst and Wehenkel [11], a scenario tree is generated from a root node (associated with an initial observation), and for each possible outcome of the problem's random variables, a child node is attached to the tree. In a multistage schema, this branching process is repeated for every child node, until a terminal state is reached. This tree generation, as described, is impossible when we're dealing with continuous random variables, and so, there's the need to discretize them.

Scenario tree generation and discretization is an important part of MSP modeling, which leads to several problems when dealing with too many variables or long time spans, since the number of necessary scenarios grows exponentially. Departing from all the approaches described above, our proposed ALM model considers a continuous state space, which naturally fits the structure of the random variables we're dealing with.

III. Reinforcement Learning

A. Markov Decision Processes

According to Sutton and Barto [1], a Markov Decision Process (MDP) is "a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states". The elements of such formalization are: a *state space* $\mathcal{S}$, an *action space* $\mathcal{A}$, an initial state $s_1 \in \mathcal{S}$ with density $p_1(s_1)$, a *reward function* $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and a *transition distribution* with density $p(s_{t+1}|s_t, a_t)$, which must satisfy the first-order Markov property: $p(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = p(s_{t+1}|s_t, a_t)$. A *policy function* is defined, to select the actions from a given state. This policy might be stochastic (defined by $\pi_\theta(a_t|s_t)$) or deterministic (defined by $a_t = \mu_\theta(s_t)$). In both cases, $\theta \in \mathbb{R}^n$ is the parameter vector of the function.

B. Basic Concepts

Reinforcement learning algorithms work through repeated interactions between the *agent* (which represents the decision maker) and an *environment* (which simulates the problem at hand). The agent observes a state s_t, and trough its current policy, takes an action a_t. Based on this state-action pair, the environment returns a reward signal r_{t+1}, and a new state s_{t+1}. The process is repeated until a terminal state is reached. The sequence $s_0, a_0, r_1, s_1, a_1, r_2, \ldots$ generated trough these interactions is called a *trajectory*.

The *return* of a trajectory is the sum of its rewards. Usually, a *discount factor* $\gamma \in (0, 1)$ is used, to avoid problems with infinite trajectories, and to enforce the fact that immediate rewards are more valuable than future ones. So, the return (or *cumulative discounted reward*) G of a trajectory τ at instant t is given by $G_t(\tau) = \sum_{t=1}^{T} \gamma^t r_t$.

Another important concept is that of *value functions*. The value function of state s_t with relation to policy π is given by $V^\pi(s_t) = \mathbb{E}_\pi(G_t|s_t)$ (the expected cumulative discounted reward of state s_t, given policy π is followed). A similar concept is the *action-value function* of state s_t, action a_t and policy π: $Q^\pi(s_t, a_t) = \mathbb{E}_\pi(G_t|s_t, a_t)$ (expected cumulative discounted reward of state s_t, assuming the agent takes an arbitrary action a_t **now**, and follows policy π **afterwards**). At last, the *advantage function* $A^\pi(s_t, a_t)$ is given by $Q^\pi(s_t, a_t) - V^\pi(s_t)$, and represents how much you can expect as an excess return

if you take arbitrary action a_t now instead of following policy π (the advantage of action a_t).

The goal of a RL algorithm is to find the policy which maximizes the expectation of cumulative discounted reward, given an initial state s_1. There are several ways of doing this, which can be divided in three major groups:

- **Tabular Methods**: better suited for discrete state and action spaces (or spaces small enough to be discretized). Approximate value functions are represented as arrays or tables, and their optimal values are calculated trough repeated iterations of *Bellman equations*: $V^\pi(s_{t+1}) = r_t + \gamma V^\pi(s_t)$ and $Q^\pi(s_{t+1}, a_{t+1}) = r_t + \gamma Q^\pi(s_t, a_t)$. The optimal policy is implicit: at any state, the action taken is the one which maximizes V^π or Q^π.
- **Policy Optimization**: the policy is explicitly given, as a probability distribution $\pi_\theta(a_t|s_t)$ (stochastic policy gradient) or as a function $a_t = \mu_\theta(s_t)$ (deterministic policy gradient). This approach is suited to continuous action and state spaces.
- **Actor-Critic Algorithms**: this is the most used framework nowadays, it's a combination of the previous two. Algorithms in this group present two components, an actor and a critic. The *actor* works as the agent, just like in policy gradient algorithms, while the *critic* estimates the action-value function $Q^\pi(s, a)$, used to update the actor's parameters.

C. Stochastic Policy Gradient

The first version of a stochastic policy gradient algorithm was developed by Willians [12] in 1992. It assumes a policy π_θ, and tries to update its parameter vector $\theta \in \mathbb{R}^n$ in the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$, where $J(\pi_\theta) = \mathbb{E}_\pi(G_t|s_t)$. In his work, Willians proves that $\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)]$, which has become known as the *policy gradient theorem*. It's an extremely important theorem, that has been used as basis for many algorithms. The first of them, proposed by Willians himself, is REINFORCE (also known as Monte Carlo Policy Gradient). At every iteration, the algorithm samples a full trajectory τ (by running the current policy), calculates $\nabla_\theta J(\pi_\theta)$ by using sampled $G(\tau)$ as an estimator for $Q^\pi(s, a)$, and then updates $\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$ (where α is a learning rate hyperparameter). Besides its theoretical importance, this algorithm is not very useful in practice, since the estimation of $Q^\pi(s, a)$ with a single rollout implies in extremely high variance, which leads to convergence issues.

D. Actor-Critic

The main idea behind actor-critic algorithms is to use policy gradient framework described previously, with a *second* function approximator to estimate $Q^\pi(s, a)$, so, avoiding the convergence issues of REINFORCE. It's like having an actor (the agent) making decisions, and a critic (the action-value function estimator) telling it how good (or bad) these decisions are. At every iteration, both actor and critic parameters are updated.

E. Deterministic Policy Gradient

In 2014, Silver et al. [13] presented their *deterministic policy gradient* theorem. Until then, it was believed that a deterministic policy gradient did not exist in a model-free algorithm, but they proved that it exists, and is the expected gradient of the action-value function. So, if $a_t = \mu_\theta(s_t)$, we have $\nabla_\theta J(\mu_\theta) = \mathbb{E}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}]$, which is a simple application of the chain rule.

F. Deep Deterministic Policy Gradient

Until 2013, it was believed that large non-linear function approximators could not be used to estimate value or action-value functions, due to several problems. One of them is dependency between sequential states; another is the change in data distribution as the algorithm is updated.

To address these problems, Mnih et al. [14] proposed an altered version of the Q-learning algorithm [15]. They introduced an *experience replay buffer*, where transition tuples $(s_t, a_t, r_{t+1}, s_{t+1})$ are stored and randomly sampled, to minimize correlations. They also introduced a *target Q network*, to provide consistent targets for the main network's training. The resulting algorithm reaches desired training stability, but works only with discrete action spaces.

Lillicrap et al. [2] adapt these ideas to an actor-critic framework, with a deterministic policy gradient agent, along with batch normalization, as proposed by Ioffe and Szegedy [16]. The resulting algorithm is known as *Deep Deterministic Policy Gradient* (DDPG).

DDPG makes use of a deep neural network, $\mu(s|\theta^\mu)$, as the actor, and another one, $Q(s, a|\theta^Q)$, as the critic. After randomly initializing the parameters of both networks, it creates copies of them, which are called *target actor* μ' and *target critic* Q'. These networks will be used to set targets for critic updates, and won't be trained like the main ones. Instead, their parameters will be updated slowly, in the direction of actor and critic learned parameters.

The actor then interacts with the environment for an arbitrary number of time steps, thus generating $(s_t, a_t, r_{t+1}, s_{t+1})$ tuples, and initialize the replay buffer. Since the policy is deterministic, a random noise $\mathcal{N}$ is added to $\mu(s|\theta^\mu)$, to ensure exploration. Algorithm 1 describes the pseudocode of DDPG.

IV. Proposed Model

We proposed to model an ALM problem as a Markov decision process. The basic inputs are: an asset (scalar representing the total amount to be invested); a liability flow (T-sized vector, with debt values for each time period); and the available assets (set of investments available to the decision maker), each represented by a time series with its historical returns. An index used to stochastically update liabilities[1] is added to available assets. Then, we define:

- State: a T-sized vector, which is the current liability vector, divided by current total asset, with $s_i \in \mathbb{R}^T$;

[1]Usually, inflation

Algorithm 1 DDPG Algorithm

procedure DDPG($\theta^Q, \theta^\mu, M, N, T$)
  Initialize critic network $Q(s, a|\theta^Q)$ with weights θ^Q
  Initialize actor $\mu(s|\theta^\mu)$ with weights θ^μ
  $\theta^{Q'} \leftarrow \theta^Q$ ▷ Initialize target networks
  $\theta^{\mu'} \leftarrow \theta^\mu$
  Initialize replay buffer R
  for episode $\leftarrow 1$ to M **do**
    Initialize random process $\mathcal{N}$
    Receive initial observation state s_1
    for $t \leftarrow 1$ to T **do**
      Select action $a_t \leftarrow \mu(s_t|\theta^\mu) + \mathcal{N}_t$
      Execute selected action
      Observe reward and next state
      Store $(s_t, a_t, r_{t+1}, s_{t+1})$ tuple in R
      Sample a random N-size minibatch from R
      $y_i \leftarrow r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$ ▷ Set target
      $\theta^Q \leftarrow \arg\min_{\theta^Q} \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
      $\nabla_{\theta^\mu}\mu|_{s_i} \approx \frac{1}{N}\sum_i \nabla_a Q(\cdot)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
      $\theta^{Q'} \leftarrow \rho\theta^Q + (1-\rho)\theta^{Q'}$ ▷ Update target networks
      $\theta^{\mu'} \leftarrow \rho\theta^\mu + (1-\rho)\theta^{\mu'}$
    end for
  end for
end procedure

- Action: an n-sized vector a, where element a_i is the proportion of total asset allocated to the i^{th} available investment, $a_i \in \mathbb{R}_+$ and $\sum a_i = 1$;
- Reward: binary function, which takes value 1 whenever the algorithm successfully pays current step's liability, and 0 otherwise.
- Terminal state: state where total asset is lesser than or equal to 0 (representing the situation in which the decision maker ran out of resources) or liability sum equals 0 (corresponding to the situation in which all debts are paid).

At every step, action a is multiplied by current total asset A_t, and portfolio's income vector I_t is simulated by a multivariate random normal, with mean vector and covariance matrix given by available asset's time series inputs. This represents available asset's monetary return for given time period. New total asset, A_{t+1}, is given by current total asset plus income, minus current step's liability. New liability flow is given by current liability, multiplied by chosen update index U, with its first element removed (representing current debt's payment), and a 0 appended to its end (to keep its size constant).

$$I_t \sim N(\mu, \Sigma)$$
$$A_{t+1} = A_t + A_t \cdot a \times I_t - L_t[0]$$
$$L_{t+1} = L_t[1:].\text{append}(0) \times U$$

These equations represent the transition distribution. As every variable at next step depends solely on variables at current step (and simulated results), this transition satisfies first-order Markov property.

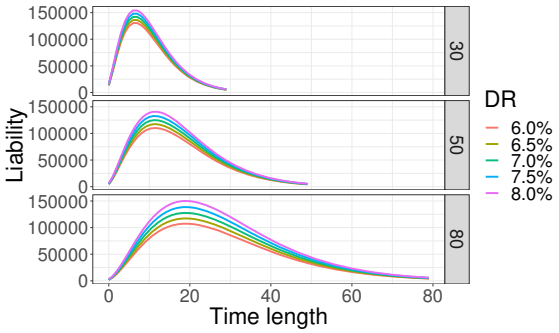

Fig. 1. Liabilities modeled as proportionate to chi-squared distributions. (a) over 30 years; (b) over 50 years; (c) over 80 years.

V. Results

A. Inputs

In order to validate our proposed model, we devised several computer simulations. We first created a gym[2] environment to test the proposed model. In such an environment, seven investments are available to the decision maker, each one representing a major asset group: IPCA (inflation), BRL-USD exchange rate, iBovespa (stocks market), IRF-M (predetermined fixed income), IMA-S (SELIC based fixed income), IMA-B 5 (short term inflation based fixed income) and IMA-B 5+ (long term inflation based fixed income). Yearly returns from 2005 to 2018 have been used.

Fifteen liability's flows have been simulated: three groups, with 30, 50 and 80 years; within each group, discount rates of 6.0%, 6.5%, 7.0%, 7.5% and 8.0% were used[3]. Flow shapes are proportional to the chi-squared distribution, as can be seen in Fig 1, to simulate the behavior of a pension fund which has not reached its maturity yet. Debts are smaller now than they will be in a near future (when more participants will retire), and after that, will slowly shrink to zero. Chosen shape is arbitrary, and other options could be used.

B. Experimental settings

The actor is a fully-connected deep neural network, with input layer the size of the respective liability (30, 50 or 80). Two hidden layers have 400 and 300 neurons each (both with ReLU activation). The output follows a softmax pattern. The critic has a similar structure, but with input size $action+state$ (i.e., 7 plus 30, 50 or 80), and a scalar output, with identity activation. Target actor and critic networks, by design, follow the same configuration of their counterparts.

Action's noise parameter introduced in training, to enforce exploration of the state space, is a vector multiplied by

[2]An OpenAI python package which provides tools to create custom reinforcement learning environments

[3]Greater discount rates means investments need better performances

actor's output. An additive parameter would lead to negative values, which is out of action's domain. It is distributed as a multivariate normal, with mean vector one, and covariance 0.01 times an identity matrix (higher values would cause too much disturbance, making the training more erratic). Resulting action is then normalized. Although more testing is needed, we suspect that the relatively high variance observed during training is mainly due to this parameter.

Reward's discount factor γ was set to 0.99; learning rate for actor and critic networks gradient updates are 10^{-3}; ρ parameter for target network's updates is 0.995; and batch size is 100. Replay buffer maximum size is 10^6, and its first ten thousand step's actions are completely random, following a dirichlet distribution with all parameters equal to 1 (multivariate version of a standard uniform distribution, which was used in the original code). In each simulation, 500 epochs have been run, with a total of five thousand steps per epoch. All the hyperparameters, except actions and state's sizes, and noise distribution, are default values of OpenAI Spinning Up (https://spinningup.openai.com/en/latest/) implementation of DDPG, which served as base for our code.

Training sessions were run in a personal computer, equipped with an Intel Core i7 7600 processor and 16Gb of RAM, running Ubuntu 18.04 and Python 3.6. A single training took about four hours to complete, and five concurrent training sessions took up to fifteen hours. The complete code and result logs can be found at https://github.com/MLRG-CEFET-RJ/DRL-ALM

C. Outputs

Figures 2, 3 and 4 show average episode return evolution for 30, 50 and 80 years groups, respectively. It's important to emphasize these returns are calculated with the training policies, that is, the noise parameter is included in the actions taken. This fact by itself raises average return's volatility.

It's easy to see that all 30 years simulations converged with no problems, and three of them did it at pretty early epochs. Besides having a very similar behavior, the 50 years group had some unstable periods, particularly for liabilities with 6.5% and 8.0% discount rates, although it was not enough to compromise final results.

As expected, the 80 years group is very unstable, when compared to the previous two. While liabilities with 6.0%, 6.5% and 7.0% discount rates converged, with a small instability for the second one, simulations with 7.5% and 8.0% had more unstable periods, and they probably should benefit from a longer training period.

D. Policy Testing

After training sessions were complete, a thousand simulations were run with each policy. In these simulations, policy is deterministic - noise parameter is no longer used. Results can be seen in table I. In each line, we have the percentile of simulations with maximum return, the minimum return, average return among all simulations and average return among simulations which did not reach maximum return.

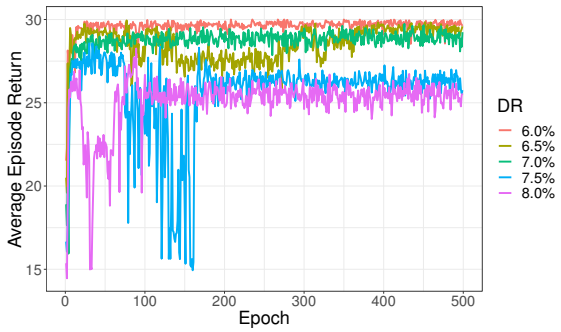

Fig. 2. Average episode return for 30-years simulations

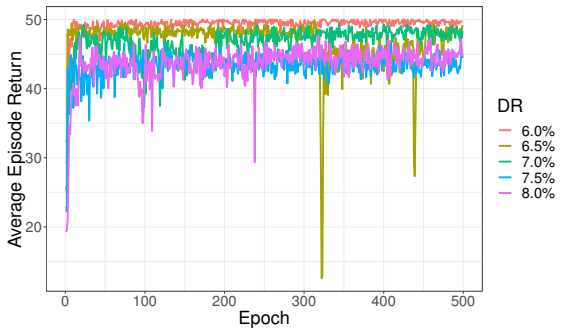

Fig. 3. Average episode return for 50-years simulations

Percentile of maximum return is over 90% in 8 out of 15 scenarios. The worst result is 52.6%, which is still reasonable, considering an 80 years time horizon and 7.5% discount rate (our claim is based on empirical knowledge). Notice that, in all time horizon groups, the simulation with 7.5% discount rate performed worse than the one with 8.0%. This is also true for average total return, except for 30-years horizon.

Figure 5 shows the distribution of non-maximal simulations. The height of each column is the amount of simulations with given return. We can see that, for the three first discount rate levels, they are pretty flat (except for the 80-years / 6.0% discount rate). In the other two, they concentrate on worst

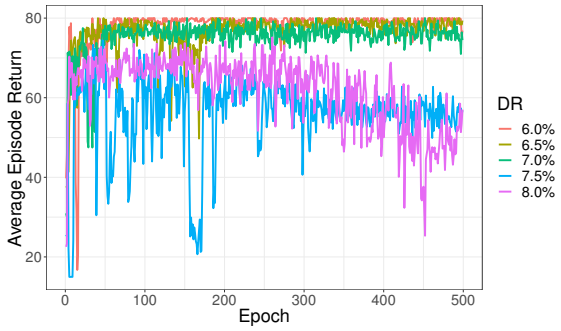

Fig. 4. Average episode return for 80-years simulations

TABLE I
SIMULATION RESULTS

Horizon	Discount Rate	% Max. Return	Min. Return	Avg. (Total)	Avg. (Not Max.)
30	6.0%	96.6%	11	29.654	19.824
	6.5%	93.9%	11	29.551	22.639
	7.0%	93.4%	9	29.199	17.864
	7.5%	67.7%	12	26.241	18.362
	8.0%	71.2%	7	25.408	14.056
50	6.0%	99.0%	25	49.815	31.500
	6.5%	93.3%	16	48.530	28.060
	7.0%	92.4%	12	48.166	25.868
	7.5%	77.7%	11	43.370	20.269
	8.0%	84.7%	6	45.118	18.092
80	6.0%	84.2%	24	76.611	58.551
	6.5%	94.6%	16	78.113	45.056
	7.0%	94.2%	17	77.278	33.069
	7.5%	52.6%	17	57.056	31.595
	8.0%	70.1%	12	64.832	29.271

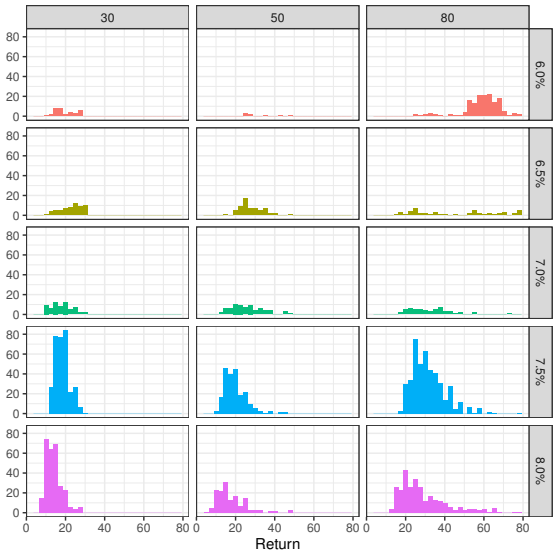

Fig. 5. Non-maximal simulations

reward levels.

One of the most important decisions when modeling a problem for a reinforcement learning algorithm is how to set its reward function, since it defines what will be optimized. Present work maximizes the amount of time periods in which the decision maker will be able to honor his debts. According to this principle, we consider the results are very good.

VI. CONCLUSIONS AND FUTURE WORKS

In this paper, common concepts of Asset-Liability Management have been introduced, and fifteen variations of a basic formulation have been solved, using an algorithm known as Deep Deterministic Policy Gradient. Results show that deep reinforcement learning algorithms are a reliable alternative to the more usual multistage stochastic programming approach, with an important advantage: there is no need to use scenario discretization.

This present work is just a first approach at Deep RL for ALM problems, and for sure, there is plenty of room for improvement. In the environment side, the next natural step would be simulating stochastic liabilities, in values and time horizons, and trying different shapes, like exponential or uniform. A change in the reward function to handle short term liquidity is also interesting. Reallocation costs would bring the model closer to reality, as well as the addition of regulatory constraints. A more sophisticated model for investment's returns is a good addition too.

For the algorithm, the reward function should be adjusted to maximize total asset's amount at the end of the episode. A stochastic policy gradient algorithm can be used to estimate investment's reallocation bandwidths, and a routine to find the optimal set of hyperparameters is definitely needed. The whole RL literature has plenty of other algorithms which can address this problem, and we also intend to test them.

REFERENCES

[1] R. S. Sutton and A. G. Barto. "Reinforcement learning: an introduction", 2nd edition. The MIT Press, 2018.

[2] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa and D. Silver. "Continuous control with deep reinforcement learning". ICLR 2015.

[3] G. P. Brinson, L. R. Hood and G. L. Beebower. "Determinants of portfolio performance", Financial Analysts Journal, v42, pp. 39-44, 1986.

[4] S.P. Bradley and D.B. Crane. "A dynamic model for bond portfolio management". Management Science, 19(2):139-151, 1972.

[5] D.R. Carino, T. Kent, D.H. Myers, C. Stacy, M. Sylvanus, A.L. Turner, K. Watanabe, and W.T. Ziemba. "The Russell-Yasuda Kasai model: An asset-liability model for a Japanese insurance company using multistage stochastic programming". Interfaces, 24(1): 29-49, 1994.

[6] D.R. Carino, D.H. Myers, and W.T. Ziemba. "Concepts, technical issues, and uses of the Russell-Yasuda Kasai financial planning model". Operations Research, 46(4):450-462, 1998.

[7] P. Hilli, M. Koivu, T. Pennanen, and A. Ranne. "A stochastic programming model for asset liability management of a Finnish pension company". Annals of Operations Research, 2005.

[8] D. M. Valladao and A. Veiga. "Optimum allocation and risk measure in an asset-liability management model for a pension fund via multistage stochastic programming and bootstrap". International Conference on Engineering Optimization, Rio de Janeiro, 2008.

[9] W. K. K. Haneveld, M. H. Streutker, M. H. Van der Vlerk. "An ALM model for pension funds using integrated chance constraints". Annals of Operations Research, 2010.

[10] N. Gulpinar, D. Pachamanova. "A robust optimization approach to asset-liability management under time-varying investment opportunities". Journal of Banking & Finance, v. 37, pp. 2031-2041, 2013.

[11] B. Defourny, D. Ernst, L. Wehenkel. "Multistage stochastic programming: A scenario tree based approach to planning under uncertainty". Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions. Hershey, Pennsylvania, USA: Information Science Publishing.

[12] R. J. Willians. "Simple statistical gradient following algorithms for connectionist reinforcement learning". Machine Learning, vol.8, pp. 229-256, 1992.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller. "Deterministic policy gradient algorithms". 31st ICML, 2014.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonouglu, D. Wierstra and M. Riedmiller. "Playing atari with deep reinforcement learning". NIPS Deep Learning Workshop, 2013.

[15] C. H. Watkins and P. Dayan. "Q-learning". Machine Learning, vol. 8, pp. 279292, 1992.

[16] S. Ioffe and C. Szegedy. "Batch normalization: accelerating deep network training by reducing internal covariate shift". Proceedings of the 32nd ICML, vol. 37, pp. 448.456, 2015.