# Policy Based Reinforcement Learning, Leiden University

**Ivo Schols** [*1] **Nikita Zelenskis** [*1]

## Abstract

In this paper, we explore the performance of REINFORCE and Actor-Critic reinforcement learning algorithms within the catch environment. Where the Actor-Critic employs bootstrapping and baseline subtraction techniques. The experimental process was split into two parts. In the first part we analyzed the effects of diverse learning rates and discount factors on the agents performance to determine the best agent. In the second part the best agent was tested in different environment settings. Results showed that the Actor-Critic agent with bootstrapping and baseline subtraction outperformed its counterparts, showing the quickest convergence and the highest win rate.

## 1. Introduction

In this report we look at two policy based agents and how they operate in the Catch environment. This report is made for the course Reinforcement Learning. Originally Catch was released in the deepmind bsuite (noa, 2023), however it has been adapted by course instructor Thomas Moerland for the policy based assignment on which this paper is based.

The default Catch environment consists of a $7 \times 7$ grid, where the agent controls a paddle in the bottom row (i.e., the paddle can only move horizontally) and tries to catch a falling ball. Either by moving left to right, or by standing still. Balls can only move vertically from top to bottom, there is no side to side movement. Balls drop randomly from one of the 7 columns at the top and have a speed of 1 by default. Meaning that only one ball is present in the environment. A catch is marked as failed when the ball hits the ground (i.e., is on the bottom row). A reward of $+1$ is given when a successful catch is made, that is both the ball and paddle occupy the same position at a given timestep. A penalty of $-1$ is given when the catch is a fail. A reward of $0$ is given in all other situations. The environment terminates in two cases. In the first case the max amount of steps, $250$, is reached. In the second case, the maximum amount of misses of $10$ is hit.

Various settings can be altered in the environment, such as: the row count, column count, ball drop rate (speed), the max amount of steps, the max amount of misses and the observation type. Observation type determines how the state of the environment is fed into the neural network. It can be encoded in two ways: vector and pixel. In the vector encoding, a vector of length three is returned with the x-position of the paddle, the x-position of the lowest ball and the y-position of the lowest ball. The second encoding is a 2d array, where the first array contains the position of the paddle in x- and y-positions and the second array contains the x- and y-positions of the falling balls.

Two on-policy gradient learning algorithms are made. Specifically, they are: REINFORCE and Actor-critic. Gradient-based learning algorithms use the derivative of the objective function to update their policy, which in this case are the weights of their neural networks. Both algorithms use Monte Carlo sampling to determine the probability that an action will result in the highest expected reward, based on the objective function, denoted as $J(\theta)$, where $\theta$ equals the parameters of of the neural networks of the agents. The background section 2 will go into more depth on REINFORCE, Actor-critic and on how to calculate the derivative for gradient ascent, which is used by both to update their network weights.

As stated, the algorithms are on-policy. On-policy means that the policy is not only used to collect data, but that all the data that the policy produces is also used to update the policy itself. This is in contrast to off-policy algorithms, were the collected data might not be used to update the policy of the agent. But instead, for example, the most optimal recorded value is used to update the policy. On-policy can be slow to converge as it not always uses the most optimal value to update the policy. However the agent is more likely to adapt to a changing environment, as it relies on the most recent data to update its policy instead of the most optimal action for a given state.

[1]Leiden Institute of Advanced Computer Science (LIACS), Leiden. Correspondence to: Schols Ivo <i.w.schols@umail.leidenuniv.nl>, Zelenskis Nikita <n.zelenskis@umail.leidenuniv.nl>.

*Proceedings of the $38^{th}$ International Conference on Machine Learning*, PMLR 139, 2021. 

## 2. Background

### 2.1. Log-derivative trick

As denoted in the introduction section 1, the Monte Carlo policy gradient is used to update the policy or rather the objective of the agents, to help it find the optimum value. Thomas Moerland introduces the following identity in his lecture notes, equation 1 (Moerland). Where $\nabla$ is the gradient, $h_0$ is a full trace, $\mathbb{E}_{h_0 \sim p_\theta(h_0)}[R(h_0)]$ the expected value of the discounted returns of full traces.

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{h_0 \sim p_\theta(h_0)}[R(h_0)] \tag{1}$$

However this equation is cannot be calculated as is. The gradient should be moved inside the expectation, to be able to compute the gradient of the expected value under the given parameters. In the same document, the log-derivative trick is shown in section 3, equation 2. Applying this trick reveals the following equality.

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)}[R(h_0) \cdot \nabla_\theta \log p_\theta(h_0)] \tag{2}$$

This expression can further be reduced to find the log-derivative of a trace distribution. Further alterations are made to be able to use this expression in practice. The function is converted to a loss function which can be used to update the weights of the agents neural network. As well as the multiplication with $-1$, since we are now minimizing instead of maximizing (i.e., we want the smallest loss possible). Where $M$ is the amount of traces calculated, $n$ the max length of a trace and $\pi_\theta(a_t \mid s_t)$ the parameterized policy returning an action $a$ at timestep $t$, given state $s$.

$$L(\theta) = -\frac{1}{M} \sum_{i=1}^{M} [\sum_{t=0}^{n} R(h_t^i \log \pi_\theta(a_t \mid s_t)] \tag{3}$$

### 2.2. REINFORCE

REINFORCE (Williams, 1992) is a Monte Carlo policy gradient algorithm. REINFORCE uses Monte Carlo to generate sample traces, which in turn are used to estimate the expected return. Sampling allows the policy to answer which action, given the current state, is most likely to yield the highest reward. As is standard with Monte Carlo Tree Search algorithms, increasing the amount of traces sampled increases the accuracy, with diminishing returns, at the cost of increased computation time.

The REINFORCE algorithm implemented uses the loss function from equation 3. However two changes have been made to improve the performance of the algorithm. Returns are normalized by calculating their z-score. Reducing variance and helping convergence. Additionally, entropy regularization is added. A more thorough explanation is given in section 2.4.

### 2.3. Actor-Critic

Actor-Critic (Konda & Tsitsiklis, 1999) is an algorithm that is similar to REINFORCE. Actor-Critic consists of two components: the actor and the critic. The actor is responsible for learning the policy and the critic estimates the value function.

The critic value function is updated by the following formula (Moerland):

$$L(\phi|s_t, a_t) = (\hat{Q}_{MC}(s_t, a_t) - V_\phi(s_t))^2 \tag{4}$$

In this case $\hat{Q}_{MC}(s_t, a_t)$ is the discounted reward from the episode and $V_\phi(s_t)$ is the value function approximation given by the neural network. $\hat{Q}_{MC}(s_t, a_t)$ is computed by $\sum_{i=0}^{\infty} \gamma^i \cdot r_i$(Moerland). In theory all possible traces should be sampled. In practice however this is not possible. Therefore a small amount of $T$ traces are sampled. Resulting in higher variance. This is due to a worse approximation of the population distribution, when the trace count is low when compared to a higher trace count.

Meanwhile, the actor is responsible for learning the policy by mapping states to the optimal actions. The actor is updated by the following formula(Moerland):

$$\hat{Q}_{MC}(s_t, a_t) \cdot \nabla_\theta log \pi_\theta(a_t|s_t) \tag{5}$$

Where $\pi_\theta(a_t|s_t)$ is the current optimal policy $s_t$ given action $a_t$.

#### 2.3.1. Bootstrapping

MC policy gradient is unbiased however, it does suffer from high variance, which can be reduced by bootstrapping. Bootstrapping gives the ability to create a trade of between bias and variance. This is done using an estimate of the future returns, instead of relying solely on the actual return from the complete episode. To update critic value function term $\hat{Q}_{MC}(s_t, a_t)$ can be substituted with $\hat{Q}_n(s_t, a_t) = \sum_{i=0}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\phi(s_n)$ where $n$ is the number of steps for the target(Moerland).

#### 2.3.2. Baseline subtraction

Baseline subtraction is the second approach that is implemented to reduce bias. Given a set of rewards for some actions, their values are subtracted by the expected reward of the critic agent. This is especially useful when, for example (taken from (Moerland), the sampled actions return 65, 70 and 75. Normally the agent would see these values as relatively the same. Because they are all higher than 0 and in the same range. However, we want to learn the agent to learn the difference between these rewards. Therefore, the baseline is subtracted, e.g. 70, so that the values become

$-5, 0$ and $5$. The policy will now favor the action of higher rewards and the actions of lower rewards will no longer be taken into account. Baseline subtraction normalizes returns of the action. This way, the best action is being rewarded and the worst action is being punished. Baseline subtraction can be implemented by substituting $\hat{Q}$ with $\hat{A}_{MC}(s_t, a_t)$. using the following equation (Moerland):

$$\hat{A}_{MC}(s_t, a_t) = \sum_{i=0}^{\infty} \gamma^i \cdot r_i - V_\phi(s_t) \tag{6}$$

Combined with bootstrapping the whole formula for computing the (advantage) reward looks like this(Moerland):

$$\hat{A}(s_t, a_t) = \sum_{i=0}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\phi(s_n) - V_\phi(s_t) \tag{7}$$

### 2.4. Exploration and entropy regularization

Both REINFORCE and Actor-Critic return an action probability from their policy given a state. Meaning that the policy is non-deterministic, a different action can be chosen given the same state using the same policy when the policy is called multiple times. This is beneficial in the sense that it ensures exploration, there is no need to for a selection mechanisms that ensure that the state space is explored. Additionally, the agent is more adept to changing environments. As it will continue to choose different actions (with a small probability) for a given state, even when it has converged. However there are two drawbacks, one is that the method is sample heavy, many samples are required to establish the proper state-action distribution. Two is the so-called collapse of the probability distribution. In this case the distribution of the policy has become narrow and heavily favors certain actions, it no longer explores different actions. Making the agent susceptible to two phenomena. One is that the agent no longer adapts to a changing environment and two the agent gets stuck in a local optima. Preventing this is done by adding entropy to the expected reward objective function ($\nabla_\theta J(\theta)$). Entropy is calculated by taking negative of the sum of action probabilities times their logarithm, the general formula is shown in equation 8(Moerland).

$$H(X) = -\sum_x P(x) \cdot \log P(X) \tag{8}$$

## 3. Methodology

As explained in the previous sections, this paper will look at the performance of two algorithms, namely REINFORCE and Actor-Critic with bootstrapping and baseline subtraction. However for the Actor-Critic algorithm an ablation study is conducted. So that in total four algorithms are tested (REINFORCE, AC w/ bootstrapping, AC w/ baseline subtraction and AC w/ bootstrapping and baseline subtraction). Experiments consist of two parts.

In the first part, the performance of the four algorithms is evaluated in the default Catch environment ($7 \times 7$ shape, speed$=$ $1.0$, max steps$=$ $250$, max misses$=$ $10$ and observation type$=$pixel). We compare performance of the four algorithms using different learning rates ($\alpha \in \{0.1, 0.01, 0.001\}$) and different discount factor values ($\alpha = 0.001$, $\gamma \in \{0.9, 0.99, 0.999\}$). The best performing agent from part one, is used in the second part of the experiments.

The second part consists of the evaluation of the best agent with different environment settings. The Cartesian product of different environment shapes and speed settings is evaluated. Where shape$=$ $\{(7\times7), (14\times7), (7\times14), (14\times14)\}$ and speed$=$ $\{0.5, 1.0, 2.0\}$. After these experiments the last experiment is ran where the agent performs in a $7 \times 7$ grid, speed$=$ $\{1.0, 2.0\}$ and observation type$=$ vector.

### 3.1. network architecture

#### 3.1.1. REINFORCE

The REINFORCE algorithm network in this paper is implemented as a sequential model consisting of a linear layer of size state space times hidden size, a ReLU function, a linear layer of size hidden size times actions space and a softmax function over dimension zero. The state space depends on the shape of the environment and its observation type. For observation type it can be calculated as follows state_space $=$ rows $\times$ columns $\times$ $2$. A factor of two is needed as the positions of the paddle and balls each have their own array. For observation type vector, its shape is equal to three (see section 1. The hidden size is equal to $128$ and the output size is equal to the action space, which is $3$.

#### 3.1.2. Actor-Critic

The network for the Actor-Critic algorithm consists of two neural networks. One for the actor and one for the critic. The neural network of the actor is same as the neural network of the REINFORCE algorithm.
The neural network of the critic is implemented as a sequential model consisting of a linear layer with input size of the action space and a output of the hidden layer size. From the first later it is connected via ReLU to a second linear layer with same input size as the output if the first layer, and the output is mapped to one float value.

### 3.2. Performance metric

To be able to compare different algorithms with each other there needs to be a common metric between the algorithms. One of the approaches could be to take the average or the

sum of rewards per episode. This nevertheless does not scale with the amount of traces. Lets say first algorithm takes 5 traces and the second algorithm takes 20 traces. This would mean that the reward range of the first algorithm is four times smaller than the range of the second algorithm.
A better solution is to use win rate over the episode. This is done by counting the number of balls dropped vs the number of balls caught. This allows for a performance metric that scales regardless of the parameters of the algorithm or the environment.
An other approach could have been comparing the loss functions of different algorithms. This however might not always work if loss functions are too distinct from one another.

## 4. Experiments

As denoted in section 3, experiments are divided over two parts. These are shown here. Some figures have a standard deviation whilst others have not. This is due to a fault in collecting experiment data.

### 4.1. Part 1

Figure 1 shows the first set of experiments for the four agents with three different learning rates.

From the first set of experiments it is clear that a learning rate of $0.0001$ results in the agents not learning at all. A learning rate of $0.001$ shows convergence with some variance around the mean. Last but not least, a learning rate of $0.01$ is optimal for convergence and having low variance between runs. Of experiments with $\alpha = 0.01$, the order of convergence, from quickest to slowest is: AC w/ bootstrapping and baseline subtraction, REINFORCE, AC w/ bootstrapping and AC w/ baseline subtraction.

Figure 2 shows the second set of experiments for the four agents, with $\alpha = 0.001$ and three different discount factors.

The second set of experiments of part 1 show that all four agents perform best with $\gamma = 0.9$, and with a closely following $\gamma = 0.999$. $\gamma = 0.99$ is quick to learn however, it also quickly stagnates, where the other two discount factor values still improve. In the agent comparison at the bottom of the figure, AC with baseline subtraction eventually performs the best. With a close finish between REINFORCE and AC with bootstrapping and baseline subtraction. AC with only bootstrapping performs the worst at all times.

### 4.2. Part 2

Figure 3 shows the first set of experiments for the Actor-Critic agent with bootstrapping and baseline subtraction, with $\alpha = 0.01$ and $\gamma = 0.9$, in differently shaped environments and different speed values.

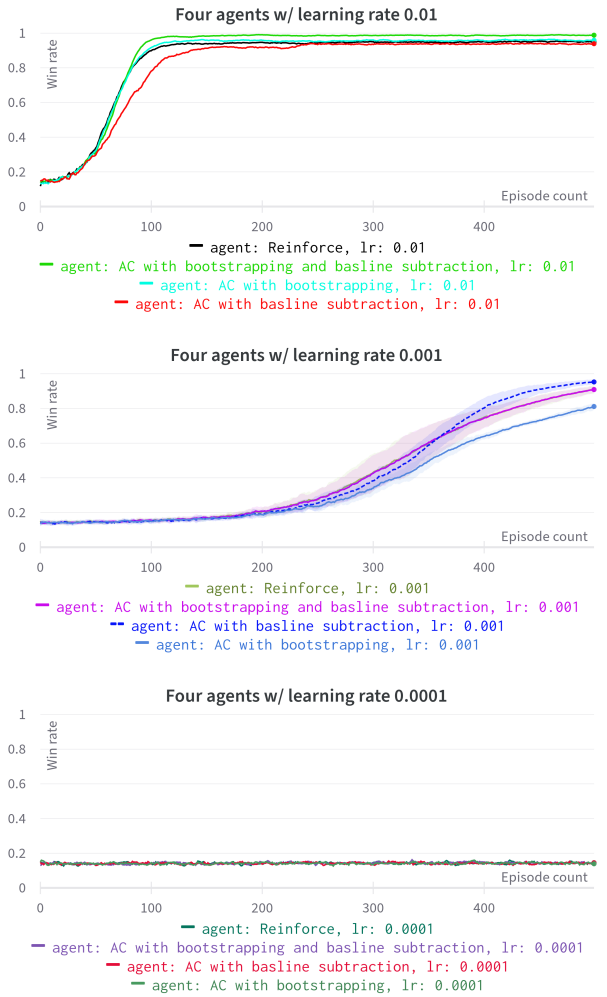


*Figure 1.* Four agents are compared with different learning rates: REINFORCE, AC w/ bootstrapping, AC w/ baseline subtraction and AC w/ bootstrapping and baseline subtraction, with $\alpha \in \{0.01, 0.001, 0.0001\}$. Experiments are ran 15 times. The standard deviation is shown as a lighter color around the mean. Exponential moving average smoothing has been added with value 0.5. Episode count length is 500.

Figure 4 shows the second set of experiments for the Actor-Critic agent with bootstrapping and baseline subtraction, with $\alpha = 0.01$ and $\gamma = 0.9$. The environment has shape $7 \times 7$, and observation type vector. Balls drops with speed 1.0 and 2.0. The agent did not finish all episodes, a count of 500, but was cut off earlier due to distribution collapse.

## 5. Discussion and further research

The first part of the experiments was ran all in one batch over night. This led to non optimal experiments as experiments for calibrating $\gamma$ did not have the best learning rate. The

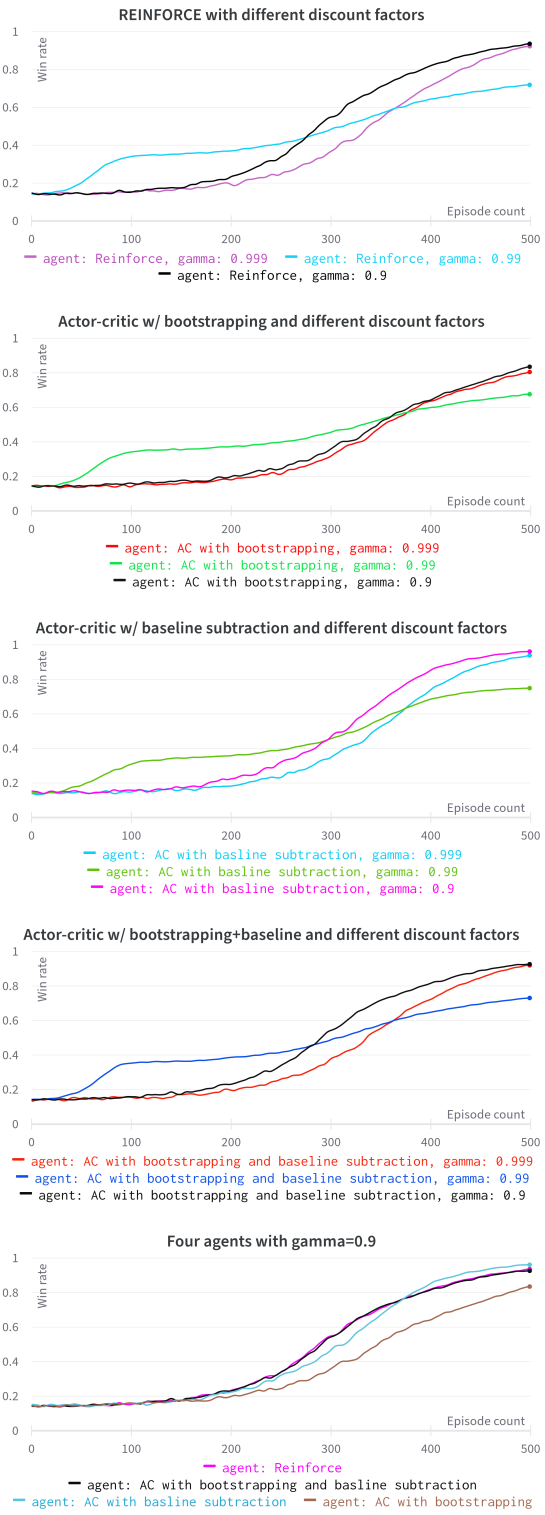


*Figure 2.* REINFORCE, AC w/ bootstrapping, AC w/ baseline subtraction and AC w/ bootstrapping are compared with one learning rate and different gamma values, with $\alpha = 0.001$ and $\gamma \in \{0.9, 0.99, 0.999\}$. Finally agents are compared for $\alpha = 0.001$ and $\gamma = 0.9$. Experiments are ran 15 times. The standard deviation is shown as a lighter color around the mean. Gaussian smoothing has been added with value 5. Episode count length is 500.

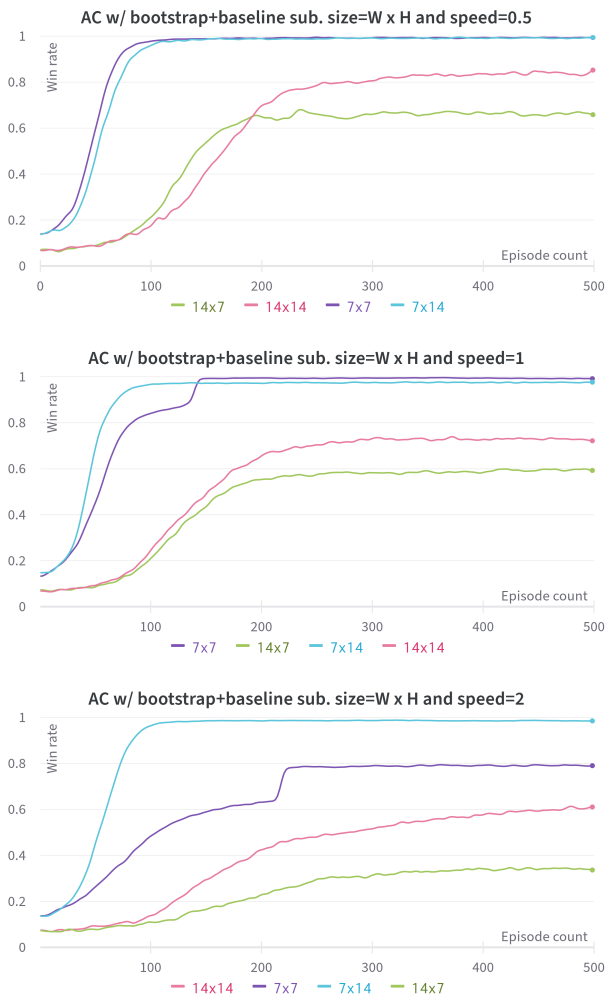


*Figure 3.* Actor-Critic agent with bootstrapping and baseline subtraction is compared in differently shaped environment sizes and with different speed values. The agent has hyperparameters $\alpha = 0.01$ and $\gamma = 0.9$. Experiments are ran 15 times. The standard deviation is shown as a lighter color around the mean. Gaussian smoothing has been added with value 3. Episode count length is 500.

learning rate was $0.001$ instead of $0.01$. In further research calibrating gamma could be fixed.

## 5.1. Part 1

### 5.1.1. Learning rate

From the first set of experiments it is evident that an $\alpha$ of $0.0001$ is not sufficient for convergence in this problem. The agent did not converge, with $\alpha = 0.0001$ for over 500 episodes. This was probably due tot the sampling sparsity. On-policy agents generally need more samples before converging to the optimal solution. With more episodes it

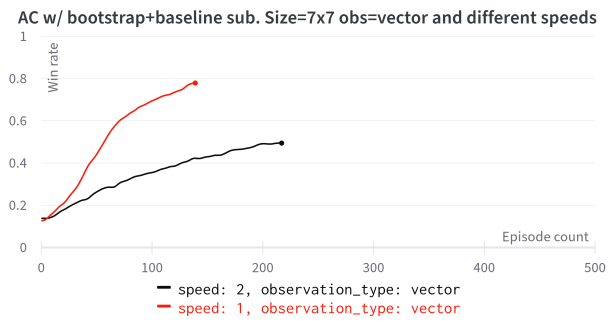


*Figure 4.* Actor-Critic agent with bootstrapping and baseline subtraction is compared in a $7 \times 7$ environment, observation type vector with different speed values (1.0 and 2.0). The agent has hyperparameters $\alpha = 0.01$ and $\gamma = 0.9$. Experiments are ran 15 times. Gaussian smoothing has been added with value 3. Episode count length is 500.

might have started converging. An $\alpha$ of $0.001$ does converge, however it takes more training episodes compared with $\alpha$ of $0.01$. Besides faster convergence it also does not seem to have any added instability. Which might be expected for higher learning rates. From the experiments it follows that $\alpha = 0.01$ seems to be the optimal choice. In further research, experimenting with higher $\alpha$ values might be insightful to see if the agent is also stable enough with these learning rates, e.g. experiments with $\alpha \geq 0.1$.

#### 5.1.2. Discount factor

All four agents are tested and compared with the three defined discount factor values, $\gamma \in \{0.9, 0.99, 0.99\}$. For all four agents it is notable that $\gamma = 0.99$ learns quicker but performs worse in the end when compared with the higher and lower discount factor values. This can probably not be attributed to getting stuck in a local optima, as the higher discount factor does seem to converge. Increasing the amount of episodes might reveal that $\gamma = 0.99$ does eventually converge, despite having been stuck in a local optima.

The highest discount factor was surprising, as it was expected to quickly learn but then no longer unlearning its short term approach for the long term optimal approach. It could also be that for this environment is pays off to be greedy. To always go to the lowest falling ball.

As expected the discount factor of $0.9$ performed the best. It is not low enough so that the agent forgets all its learned behaviours before convergence but it is not high enough so that the agent might get stuck in its old ways. Additionally, a discount factor of $0.9$ allows the agent to more easily adapt to changing environments when training. This can be useful when the agent learns in a more sparse environment (i.e. with a greater size, or more balls dropping). It is more likely to reduce the importance of rewards it has already seen and more easily adapt to new stimuli (e.g., balls dropping at the end of the room).

#### 5.1.3. Summary

In summary $\alpha = 0.01$ and $\gamma = 0.9$ had the best performance for all agents. Moreover AC agent with bootstrapping and baseline subtraction seems to have the fastest convergence, while also maintaining a higher win rate compared to all other tested agents.

### 5.2. Part 2

The results of first set of the experiments highlight that none of the agents could converge in the environment with a $14 \times 14$ grid. This is most likely due the fact that the hidden layer size is too small for a larger grid and cannot be estimated by the network. Furthermore setting speed to $2.0$ resulted in only one environment being able to have a perfect win rate ($7 \times 14$). This is to be expected as it is not possible to catch all balls when the environment is too large for a certain speed.
Furthermore there is a bump in the $7 \times 7$ environment at episode 120 for speed 1 and episode 210 for speed 2, we do not have an explanation why this happened.

#### 5.2.1. Observation type

Setting the observation type to vector led to a slower convergence. This shows that a full state space leads to a better performance as it can predict where to catch the ball with a higher accuracy.
In further research it would be interesting to see if changing the size of the hidden layer will result in convergence on a bigger grid.

## References

Behaviour Suite for Reinforcement Learning (bsuite), May 2023. URL https://github.com/deepmind/bsuite. original-date: 2019-08-02T15:36:05Z.

Konda, V. and Tsitsiklis, J. Actor-critic algorithms. In Solla, S., Leen, T., and Müller, K. (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Moerland, T. Continuous Markov Decision Process and Policy Search. URL https:

//thomasmoerland.nl/wp-content/uploads/2021/04/continuous_mdp.pdf.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.