Deep Reinforcement Learning
– Assignment 3: Policy-Based Reinforcement Learning –

Lukas Welzel [1] **Rens Kievit** [1] **Benjamin Silk** [1]

Abstract

In this report we attempt to apply and study multiple policy based reinforcement learning methods to the cartpole environment. Using grid-like hyperparameter searches, and an ablation study we explore and discuss the performances and properties of these methods. We find that policy search approaches require outside help to increase stability, but the addition of a value-based critic to an agent already significantly boosts performance.

1. Introduction

In this assignment we study policy-based (deep) reinforcement learning. This stands in contrast to the value-based approaches we used in previous assignments: instead of an agent learning the values of actions in states, it directly learns the optimal policy.

1.1. Environment

The environment used in this report is that of a simple cartpole first described by Barto et al. (1983), and implemented by Brockman et al. (2016) on `OpenAI Gym` as `CartPole-v1`. The cartpole consists of a rectangular cart which can move along a horizontal axis, and a pole, attached with a single joint to the cart. This entire system is described by four parameters: the cart position x, the cart velocity $\dot{x}$, the pole angle θ and the pole angular velocity $\dot{\theta}$. These are all contained in the state vector s_t. The goal of the agent is to balance the pole upright ($\theta \in [-12^o, +12^o]$) for as long as possible without deviating too far from the center ($x \in [-2.4, +2.4]$). The agent interacts with the environment by applying a fixed force, either to the right or left, at each timestep to the cart. It receives a reward of $+1$ for each survived timestep, and it has "won" an episode once a total reward of 200 is achieved. For the actor-critic algorithms we further process rewards by disincentivizing actions that resulted in the premature termination of the environment by passing a large negative reward, scaled by the maximum expected reward of a converged policy, to the function computing the bootstrapped, discounted rewards for the agent.

[1]Faculty of Science, Leiden University, The Netherlands. Correspondence to: L. Welzel (A2C) <welzel@strw.leidenuniv.nl>, R. Kievit (REINFORCE) <kievit@strw.leidenuniv.nl>, B. Silk (REINFORCE) <silk@strw.leidenuniv.nl>.

2. Method

Policy based approaches, or **policy searches**, attempt to directly optimize the policy, instead of deriving an optimal policy using state action values using approaches such as Q-learning. The goal is to find the parameters θ that give the highest value in the initial state, described by the function $J(\theta)$, which is defined as

$$J(\theta) = \mathbb{E}_{h_0 \sim \pi_\theta(h_0)}\big[\mathrm{R}(h_0)\big]. \tag{1}$$

Where h_0 is the full trace, or path, taken by the agent, this is sampled from the policy π_θ characterized by the parameters θ. $\mathrm{R}(h_0)$ is the discounted reward observed by the agent along the given trace. To achieve these parameters, we make a distinction between **gradient-based** and **gradient-free** optimization methods. As the name implies, in gradient-based optimization we compute the gradient of $J(\theta)$ with respect to all the parameters θ, and apply gradient ascent to move towards the optimal parameters. This gradient, however, is not simple to compute because the dependence on θ of J is only hidden in the distribution on which the traces are sampled. The solution employed by most gradient-based policy searches is the log-derivative trick to pull the gradient into the expectation. Sparing the full derivation, this results in

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

The majority of this report will be focused on these methods, more specifically, we will explore the `REINFORCE` and `Actor-Critic` applications. Gradient-free methods don't move through the parameter space using gradients, but use other approaches such as evolutionary strategies instead. Later in this report, we will look at an application of the cross-entropy method for evolution, or `CEM`. For our numerical calculations and neural network handling we make use of the `NumPy` Harris et al. (2020) and `TensorFlow` Abadi et al. (2015) Python packages respectively.

2.1. REINFORCE

The REINFORCE method, also known as the **Monte Carlo Policy Gradient Algorithm** is one of the simplest approaches to policy searches. It simply attempts to maximize the function given in Equation 1 using its gradient, given in Equation 2. However, usually we only care about the rewards observed by the agent after a given point in time instead of using the full trace reward for all actions. Additionally, because we we are dealing with finite time and computing power we have to approximate the expectation value by averaging over multiple traces through the environment. All of this combined gives

$$\nabla_\theta J(\theta) \approx \frac{1}{\mathrm{M}} \sum_{i=1}^{\mathrm{M}} \Big[\sum_{t=0}^{\mathrm{N}} \mathrm{R}(h_t^i)\, \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) \Big] \qquad (3)$$

Where M is the amount of traces over which we average, and N the amount of timesteps in trace i. h_t^i gives the discounted, future reward of trace i from timestep t. During the sampling of traces, we allow the default agent to explore the environment with a simple exponential decaying ϵ-greedy exploration strategy, where epsilon is decreased as $\epsilon \leftarrow \epsilon^r$ with r the decay factor as provided in Table 1. In our implementation of the REINFORCE algorithm, we define a loss function $L(\theta) = -J(\theta)$, where we introduce the minus sign because we want to minimize the loss. Note the absence of the gradient in L, we first compute the total loss using the logarithmic state-action probabilities, and future discounted rewards. Then we differentiate the loss with respect to all trainable network parameters using automatic differentiation through `TensorFlow.GradientTape`, and apply gradient descent using these gradients. The calculating of $\log \pi_\theta$ is handled through `TensorFlowDistributions` to create a categorical distribution over the two possible actions from the network outputs. This means that the network updates are off-policy, as the exploration induced by our own strategy is different from what is produced to create the log probabilities. This is necessary to ensure that the gradient tape can properly follow the path of the loss function to ensure proper differentiation. The full process is given in algorithm 1. Due to the absence of any stabilizing additions in this implementation of policy search, the REINFORCE algorithm is said to be very unstable to both hyperparameter choice, and to the random initial conditions. Our aim thus is to search for good hyperparameter settings that return a proper training agent. The default settings we use are given in Table 1, which we will use as the center point of a hyperparameter search with the goal to find a relatively stable setting.

Learning Rate α	0.01
Discount Factor γ	0.80
Starting Exploration Factors ϵ	1.00
τ	1.00
Exploration Strategy	ϵ-greedy
Annealing Method	Exponential
Decay Factor r	0.9
Post-Decay Minimum ϵ	0.1
Hidden Layer Configuration	[128, 64]
Hidden Layer Activation Function	ReLU
Output Layer Activation Function	softmax
Weight Initialisation	HeUniform
Optimizer	Adam
Sampled Traces per Epoch	5

Table 1. REINFORCE: Default hyperparameters.

2.2. Actor-Critic

We implement three versions of the actor-critic agent: one with bootstrapping, one with baseline subtraction, and one with both. We aim to find the effects of bootstrapping and baseline subtraction, and their combination, within the actor-critic framework on the variance of the policy gradients. We also implement several other extensions; entropy maximization, both with constant target entropy and scheduled target entropy annealing (Xu et al., 2021), policy annealing, and attempt to implement (prioritized) experience replay (Wang et al., 2016). We perform an ablation study, where we compare the default advantage actor-critic (A2C) (see Table 2 for default hype-parameters) with A(2)C without bootstrapping and baseline subtraction. We tune the hyperparameters of the A2C based on a study of hyperparameter effects which we examine below. We test that we do not miss interesting features by limiting the maximum episode length to 200 by running an "unrestricted" environment, see Figure 14.

Use Bootstrapping	Yes
Use Baseline subtraction	Yes
Use Experience Replay	No
Anneal Policy	No
Use Entropy Maximization	Yes (constant target)
Batch Size	60
Learning Rate α	5×10^{-3}
Discount Factor γ	0.95
Exploration Strategy	rescaled $tanh$ from logits
Annealing Method	Exponential ($\delta = 0.99$)
Initial Exploration Temperature τ_0	1.0
Value func. reg. constant	0.5
Entropy reg. constant α	10^{-4}
Actor & Critic Hidden Layer Config.	(32, 16)
Actor Layer Act. Func.	$tanh$
Actor Output Act. Func.	$tanh$
Critic Layer Act. Func.	ReLU
Critic Output Act. Func.	None/linear
Weight Initialisation	Gorlot/Xavier uniform
Optimizer	RMS proportional

Table 2. A2C: Default hyperparameters.

2.2.1. Baseline Actor-Critic

AC methods separate policy and value function, called actor and critic, to independently learn to act and assess the value of actions. The critic learns on-policy by observing the rewards of the policy the actor follows. The critic thus learns a value function and outputs a TD error which both the actor and critic use to learn. (Sutton & Barto, 2018) We implement the actor and critic as a single dense neural network, which receives a state as input and then diverges into the actor and critic branch. We only consider networks for which both branches have the same architecture. The actor returns logits for each state, essentially non-normalized action probabilities from the current policy and the critic returns the state value as a single non-normalized scalar. The actor samples a categorical distribution based on the logits to chose the next action. We update the networks using batches of observations scheduled over the learning process. The networks are updated separately using a actor/policy loss function for the actor branch and a value loss function for the critic. We naively implement the critic loss function as a soft Bellman backup, i.e. half the mean-squared-error of the value function and returns of the states sampled in the batch. We use `tensorflow`s categorical cross-entropy functions to evaluate the loss under the current policy in the batch. This is equivalent to taking the negative log-probability (multi-class cross-entropy) (Bishop & Nasrabadi, 2006), at least for binaries. We then implement entropy maximization for the policy loss by evaluating the auto-cross-entropy of the re-normalized (softmax) logits themselves which is used to estimate the Kullback–Leibler divergence[1] which is then minimized by `tensorflow`. (Xu et al., 2021) The A2C we implement is then an off-policy algorithm since we maximize its entropy and then use soft updates. Keeping the loss functions internal to `tensorflow` drastically reduces computation time, anecdotally the default agent learns on 1000 episodes in less than 5 minutes with a mean episode reward of $170/200$. Below we list the components and additions of our implementation, which can be independently stacked. We decided against implementing an Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016) since 1) A3C does not have an inherently different method to A2C and draws its advantage mostly from its ability to explore larger state-action spaces and 2) using distributed agents would make our hyperparameter more study difficult since it is also parallelized.

[1] I am not actually sure if its really a direct KL divergence we minimize since we feed through layers using tanh-activation and then renormalize using softmax only in the loss function itself.

2.2.2. Bootstrapping

We implement bootstrapping as shown in Equation 4.

$$\hat{Q}_\pi\left(s_t, a_t\right) = \sum_{k=0}^{n-1} r_{t+k} + \gamma^k V_\pi\left(s_{t+n}\right). \tag{4}$$

While we do not implement an explicit time horizon, we do discount future rewards by γ^k, nevertheless, the time horizon is implicitly the batch size of the update. We account for the truncated returns after the time horizon by letting the critic predict the returns beyond the horizon. In the following we will call these bootstrapped, discounted rewards returns. (Sutton & Barto, 2018)

2.2.3. Baseline Subtraction

We implement baseline subtraction (BLS) using the (bootstrapped) returns and the value function from the critic. We will call the result of the BLS, shown in Equation 5 the advantage. By using BLS the critic is able to differentiate whether an action was good or the state was good. (Sutton & Barto, 2018)

$$A\left(s_t, a_t\right) = Q\left(s_t, a_t\right) - V\left(s_t\right) = r_{t+1} + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \tag{5}$$

The update equation, with the appropriate value function, potentially including BS, discounted rewards, is given by Equation 6.

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{a\sim\pi}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right) A_\pi\left(s_t, a_t\right)\right] \tag{6}$$

2.2.4. Maximum Entropy

One important measure of the performance of agents is the entropy of a policy. A2C agents can be exceptionally brittle and easily converge to non-optimal policies. By trading off expected returns and the entropy of a policy we can encourage exploration to prevent this lock-in because we decrease the likelihood of repeatedly exploiting the same action. Here we define the entropy of a policy as shown in Equation 7.

$$\mathcal{H} = \mathop{\mathbb{E}}_{a\sim\pi}[-\log \pi(s)], \tag{7}$$

where we can view the policy π as the probability density function from which the the actions are selected and which can again be computed using `tensorflow`s cross-entropy functions. The rewards are then adjusted to account for the entropy based exploration-exploitation trade-off by Equation 8 and the converged policy is then given by Equation 9.

$$R'_k = R_k + \alpha \mathcal{H}\left(\pi\left(\mathbf{s}_k\right)\right), \tag{8}$$

where α is the regularization/maximization parameter.

$$\pi^\star = \arg\max_\pi \mathop{\mathbb{E}}_{a\sim\pi} \sum_{k=0}^{n-1} \gamma^k \left(R_k + \alpha \mathcal{H}\left(\pi\left(\mathbf{s}_{k+1}\right)\right)\right) \tag{9}$$

The soft Bellman equation is then Equation 10.

$$\hat{Q}_\pi\left(s_t, a_t\right) = R_t + \gamma \mathbb{E}_{s_{t+1} \sim D}\left[V\left(s_{t+1}\right)\right], \tag{10}$$

with the soft state-value function V_π as in Equation 11. (Haarnoja et al., 2018b;a; Christodoulou, 2019)

$$V_\pi\left(s_t\right) = \mathop{\mathbb{E}}_{a_t \sim \pi}\left[Q_\pi\left(s_t, a_t\right)\right] + \alpha \mathcal{H}\left(\pi\left(a_t \mid s_t\right)\right) \tag{11}$$

This class of AC methods is called Soft Actor Critic (SAC) and very powerful.[2] We also attempt to implement target entropy anneal scheduling (TES-SAC) based on Xu et al. (2021), however neither the moving window mean entropy and standard distribution estimator, for trajectory i: Equation 12 and Equation 13, sufficiently converge in any of the trials to begin scheduling.

$$\hat{\mu}_i = \lambda \cdot \hat{\mu}_{i-1} + (1-\lambda) \cdot \mathcal{H}_i \tag{12}$$

$$\hat{\sigma}_i = \sqrt{\lambda \cdot \left(\hat{\sigma}_{i-1}^2 + (1-\lambda) \cdot \left(\mathcal{H}_i - \hat{\mu}_{i-1}\right)^2\right)} \tag{13}$$

2.2.5. Actor-Critic Experience Replay

We attempt to implement prioritized experience replay for A2C based on Wang et al. (2016); Horgan et al. (2018), however we were unsuccessful at properly sampling from the buffer and prioritizing trajectories.

2.2.6. Policy Annealing

Since the target entropy scheduling did no yield the desired results we implement mechanism similar to softmax to anneal the policy. In the target network we found that using a hyperbolic tangent activation was extremely effective. We decided to rescale the (potential) logit probabilities from the policy using a rescaled $tanh$ function, given by Equation 14.

$$\hat{\pi}_{s_t} = \frac{e^{2\tau^{-1}\pi_{s_t}} - 1}{e^{2\tau^{-1}\pi_{s_t}} + 1} \text{ with } \hat{\tau} = \delta\tau, \tag{14}$$

where τ is the exploration temperature, which is annealed using exponential decay after every policy update and δ is the decay rate. As $\tau \to \infty$ the policy becomes random and as $\tau \to 0$ the policy becomes greedy. Notably, the action is then still sampled from a categorical distribution.

2.2.7. Batch Annealing

Finding a good batch size is traditionally a trade-off. Small batch sizes allow for quick updating during initial learning but do not carry enough information to guarantee stability and generality when episodes become longer. The inverse is the case for large batches. Since there is no practical reason why the batch size needs to stay constant during the learning we anneal the batch size using Equation 15 after every batch update.

$$\hat{\mathcal{L}}_\mathcal{B} = \mathcal{L}_\mathcal{B} \cdot (1 + \nu\omega^T), \tag{15}$$

[2] I would even say somewhat redeeming for DRL.

where $\mathcal{L}_\mathcal{B}$ is the batch size, $0 \leq \nu$ is the scaling base, $0 \leq \omega \leq 1$ is the scaling decay and T is the episode number.

2.3. Cross-Entropy Method

The cross-entropy method (CEM) is a gradient-free, evolutionary algorithm for policy optimization. Because this approach does not make use of gradients such as we saw in the previous sections, the idea and approach are relatively simple. We create a number of agents n_{agents} with network weights sampled from a pre-defined Gaussian distribution with $\mu_1 = 0$ and $\sigma_1 = 1$ for all weights. For each of these agents, we evaluate their performance and select the best $u\%$ rounded up, which we call the **elite set**. Then, we refit the Gaussian distributions on each of the weights of the elite set separately, and repeat. Here, we apply a relatively simple fitting procedure where for each of the parameters of the network, we compute the mean μ and standard deviation σ of that parameter over all agents in the elite set. The full training process is summarized in algorithm 4.

During training of the evolutionary agents, we use an ϵ-greedy approach, with a constant $\epsilon = 0.2$. We do this because we want the elite set to contain the best performing agents each epoch. Including too much exploration could cause the parameters describing the Gaussians to move the wrong way initially. Additionally, the CEM already contains exploration through the randomization present in sampling from Gaussian distributions.

Because of its simplicity, the cross-entropy method can form a good baseline to the performance of the more complicated methods described earlier.

3. Results

3.1. REINFORCE

In an attempt to find stable configurations of the REINFORCE algorithm, we perform a hyperparameter search of a small range of selected values. We decided upon reasonable standard values, as presented in Table 1, and decide upon a few different values to experiment with. The results presented here, are by no means an extensive optimal hyperparameter search, but rather a qualitative attempt at studying the dependence of algorithm performance on the variation of certain hyperparameters.

An aspect of the REINFORCE algorithm that is of immediate note is the high variability of its learning curves. Running the same agent, with the same settings on the same environment, can result in wildly differing final performances (see Figure 1), even more so than with other deep learning based methods based on our experience. On one run an agent receives just the right initial weights to be able to improve itself rather easily and quickly to rewards above 100,

whilst the next it starts off bad, and is never able to improve itself above 10. This fact, combined with the limited amount of runs that we have been able to simulate due to limited computing time, has lead to inconsistent and inconclusive results for REINFORCE.

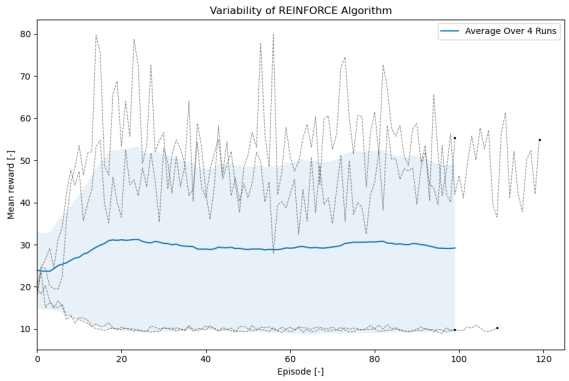

Figure 1. Four runs of REINFORCE on the same default settings (see Table 1). Although this is not apparent in this figure, the top two runs result in the agent not being able to keep the Cartpole upright once it starts falling to the left. The agent strangely does attempt to prevent falls to the right quite adequately. We suspect this to be a local minimum in which the agent is trapped.

3.1.1. Number of traces

A large portion of the variability of the REINFORCE algorithm originates from the randomness present in the environment itself in the form of the initial state. A logical way to mitigate that randomness then is through sampling a larger number of traces each epoch and thereby averaging out the randomness. In Figure 2 we present the performances of networks sampling 1, 5 and 15 traces each epoch. The instability of the REINFORCE algorithm is clearly visible when only sampling 1 trace per episode. The average performance is terrible, containing only a few peaks during periods of overfitting on certain regions in state space. Another noteworthy aspect is the terrible performance of all 4 runs with 15 traces per epoch, which shows no signs of improvement over the 200 epoch run. This is possibly due to the fact that too much uncorrelated data is received by the agent at the same time, resulting in a gradient that wants to push the weights into multiple directions at the same time, resulting in bad performance.

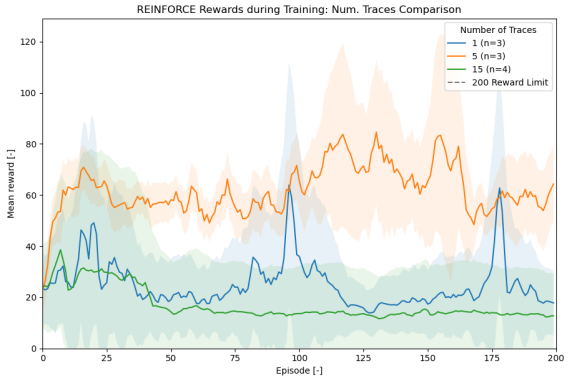

Figure 2. Study of the effect of the number of sampled traces per epoch for the REINFORCE algorithm.

3.1.2. Annealing Schedule

Due to the high variance present in the REINFORCE algorithm due to initial conditions, one would expect that it is imperative that the agent explores the state space as much as possible such that the the variance can be mitigated as much as possible through actual experiences with the environment. While we limit our study to only the ϵ-greedy exploration strategy, one avenue of study that remains is the evolution of the value of ϵ throughout the training program. In Figure 3 we present the performances of networks with an exponential-, and linear annealing schedule, and compare them to a network with an initial low ϵ without annealing. In this figure we see the best performance from the agent without any annealing schedule, while both other methods seem to learn relatively slow, or seemingly not at all. The latter is most likely an effect from the relatively short number of episodes, we already see an increasing trend in the agent with linear anneal, while the exponential anneal agent might need some time to properly explore the state space. In either case, the initial exploration factor for both is quite high, so it is also possible that what we see here is just mainly random movement, and actual performance comes when ϵ is sufficiently low.

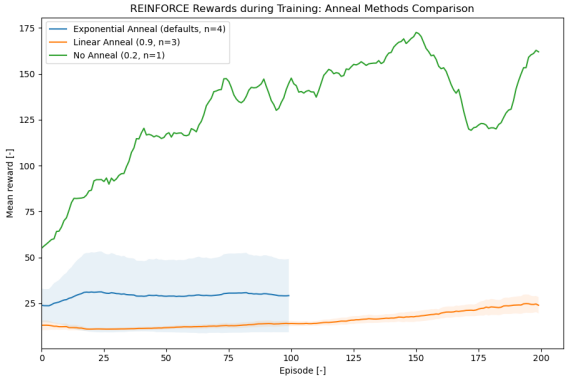

Figure 3. Study of the effect of different annealing schedules on the REINFORCE algorithm. The blue line is the same default settings run as in Figure 1. The algorithm without annealing has $\epsilon = 0.2$

3.2. Actor-Critic

3.2.1. Ablation Study

The ablation study of the AC, shown in Figure 4, was performed using tuned AC agents, hence the overall good performance. Clearly baseline subtraction is very important for AC. This is because it helps the agent find whether the action or state was good, which means it can 1) aim for high return states, 2) not get trapped making poor actions in otherwise high return states, 3) avoid being ”exploited” by inconsistencies in the environment and 4) we make smaller gradient (steps) and thus have more stable updates. In this study the effect of the BS is relatively small since the batch size is large compared to the bootstrapped trajectory. The trade-off the BS provides, i.e. reducing variance in favour of bias thus becomes less relevant. As expected, annealing the policy leads to faster initial learning, higher mean rewards and a more stable exploitation. Scheduling a batch size anneal reinforces this by taking advantage of both small batches and large batches for updating the model, when the respective advantages dominate.

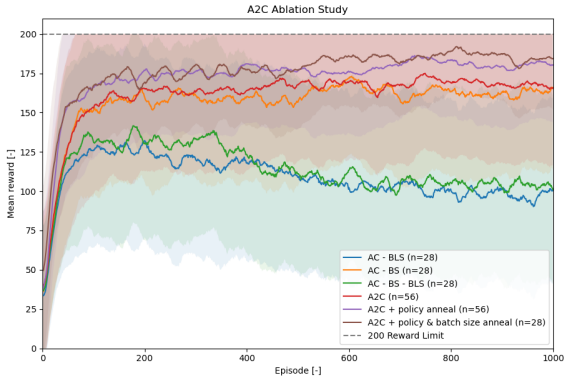

Figure 4. Ablation study of AC algorithms. The full A2C agent performs extremely well, both in terms of initial learning and mean rewards after initial learning. Removing BLS has a large impact on performance, especially when a quasi-steady state has been reached. BS has a small impact, mostly due to the size of the relatively size of the batch that is used for updating. Both of our additions improve the performance of A2C somewhat, with the full agent including policy and batch size anneal having the fastest initial learning and highest mean reward of $185/200$.

3.2.2. Hyperparameter Tuning

We tune hyperparameters mostly independently for A2C. This is insufficient to fully explore the parameter space as the performance partial derivatives w.r.t. the hyperparameters are not independent. However, covering the full parameter space is infeasible beyond some parameters which are clearly related. For each of these parameters we select reasonable initial guesses from literature (Xu et al., 2021), previous projects in the course and A2C baselines (Dhariwal et al., 2017). We then tune the system, specifically the initial performance in the first 250 steps until we find a reasonable baseline around which we study the hyperparameters. We place the plots of studies that do not have interesting results in the appendix for completeness's sake.

Batch Size & Anneal Schedule The batch size has a large impact on both the learning rate of the agent and its mean reward, see Figure 5 and Figure 6. As long as the batch contains sufficient transitions to cover more than a small region of the transition space the agent can perform well, however agents with large batch sizes naturally learn slower since each update covers a large space of (highly correlated) transitions.

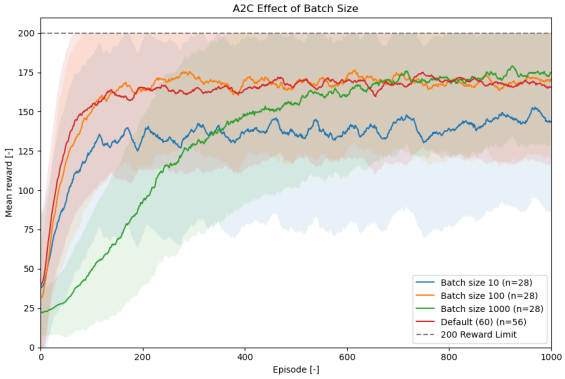

Figure 5. Study of the effect of a constant batch size for A2C. The batch size and with that the update frequency has a large effect on both the initial rate of learning and the final mean reward. For batch sizes which are much smaller than the maximum possible reward the agent is able to learn quickly but lack generality and cannot reach a high mean reward. Large batch sizes learn slowly since the initial updates sample an almost uninformed policy, however they are very stable once they sample a stable policy since they fit based on many (highly correlated!) transitions. Batch sizes that are around or slightly below

Learning Rate The impact of the learning rate is as expected, with rates around $10^{-3} - 10^{-2}$ performing well, see Figure 7. Very low learning rates can still reach high mean rewards, however they struggle to converge in a reasonable time, especially considering that entropy maximization can be used to archive a similar effect to low learning rates, however in a sample efficient way.

Discount Rate We studied the effect of the discount rate for A2C and found that, for reasonable values its effect is small, see Figure 12.

Model Architecture We constrained the study of network architectures to agents with the same branch structure for the actor and critic. This is probably not an efficient network,

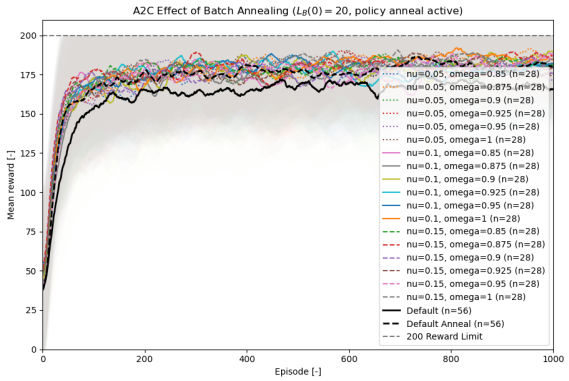

Figure 6. Study of the effect of a batch size annealing for A2C. As long as the batch size is annealed in a reasonable manner, i.e. roughly growing with the mean rewards, there is a slight benefit over not annealing the batch size. It is worth highlighting that the large batch sizes during exploitation act as a very poor replay buffer since they include multiple episodes and hence allow better generalization, however suffer from the problem of updating using correlated transitions.

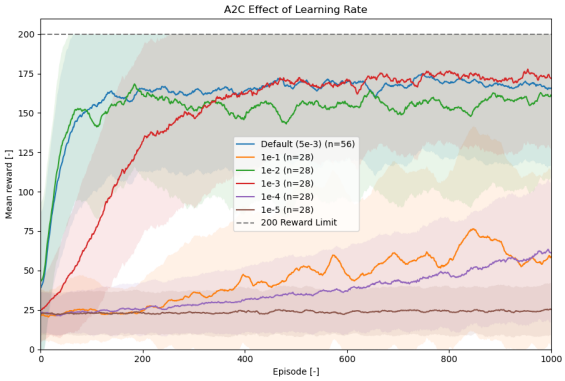

Figure 7. Study of the effect of the learning rate for A2C. Both high and low learning rates perform poorly, either by being extremely unstable or not learning sufficiently fast. Learning rates around $10^{-3} - 10^{-2}$ seem to perform best, all other parameters being equal.

since the branches of the networks have dissimilar tasks and hence dissimilar architectures can be more efficient. However, anecdotally we found that for small networks (2 layers, much less than 254 nodes per layer) the computational effort is essentially the same and the performance difference is minute. We find that even small networks accurately learn the system dynamics.

Activation Functions We performed a study of activation functions. The activation functions of the actor that accurately renormalize logits perform well, whereas the ones that do not perform poorly since they cannot properly express the networks action probabilities, see Figure 13.

Policy Anneal The annealing method we proposed works well for the problem and allows the agents to efficiently anneal from exploration to exploitation. Due to the absence of experience replay the agents cannot take full advantage of the exploration phase, see Figure 9.

3.3. Cross-Entropy Method

Because of the large amount of simulation time required for evolutionary algorithms due to the high number of required traces per generation ($n_{agents} \cdot n_{traces}$) and time constraints, the analysis here is restricted to only a single presented run, and experience from earlier iterations. In Figure 10 we present the results from a single, relatively short, run of the cross-entropy method for evolution as described in algorithm 4. Here we have chosen for a total of 40 agents, each sampling 5 traces per epoch. The number of agents is chosen to increase the amount of the parameter space that is covered each generation, and to ensure that we fit our new distributions to a number of agents to avoid overfitting towards the weights of the single best performer. Even though we only present one run here, the learning curve it shows is one we have consistently seen during testing of the code, however sadly we are unable to present these initial runs. These 16 epochs took approximately 4 hours of simulation time.

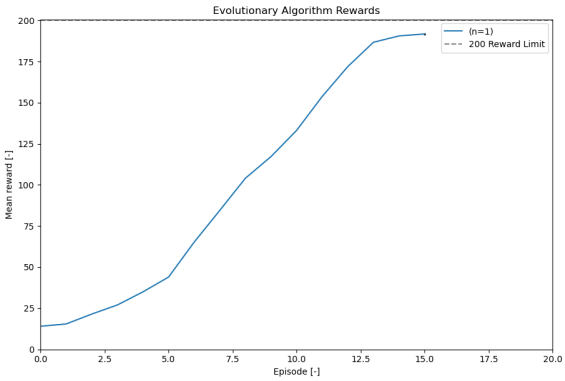

Figure 10. Single run of the evolutionary algorithm as described in subsection 2.3. Algorithm ran with 40 agents per generation, and each agent had 5 traces to average its result over. Run was stopped prematurely after plateauing for 3 generations at a mean reward of approximately 190.

We see that the performance of the algorithm grows rapidly, after 14 epochs it has already reached the maximum reward. It is remarkable how quickly and efficiently this algorithm learns, however that cannot be said without mentioning that these 14 epochs contain about as much computation as 500 epochs of any of the other methods presented here.

Figure 8. Study of the effect of the model architecture for A2C. Small, flat networks seem to perform well, in terms of stability, learning rate and mean reward. Large networks are seemingly prone to overfitting whereas even small networks can still estimate the dynamics in their model as well as medium ones, however they learn slower since the smaller size makes it more difficult to isolate the relevant features in the data.

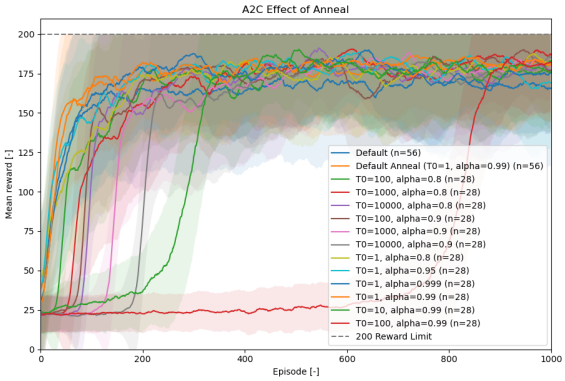

Figure 9. Study of the effect of policy anneal for A2C. Our $tanh$ anneal works reasonably well and clearly allows for the possibility to tune exploration and exploitation phases very precisely. Nevertheless, it lacks precision for anneals with large decay steps (large initial temperature and decay) due to the shape of the rescale. Anneals which explore for a long while are able to switch to very efficient exploitation quickly and reach high mean rewards quickly. The absence of experience replay makes it difficult for the agents to further exploit their exploration phase as the knowledge not kept in the updates.

4. Discussion

4.1. REINFORCE Instability

As we briefly touched on in subsection 3.1 and Figure 1, and as is apparent in nearly all of our results for REINFORCE, there is a high degree of instability and variability from run to run. Depending on the first initialization of the neural network weights, the agent either refuses to learn, or gets stuck in a local minimum with no way out. This could be due to a lack of any mechanism to steer the agent in right direction once it reaches a local minimum (for instance, a "critic"). Although this does confirm our suspicions that policy gradients used by REINFORCE indeed suffer from instability and high variance, it is unfortunate that this has resulted in our inability to study REINFORCE properly, and to build an agent that can learn and master the Cartpole environment.

These instability effects are less visible in the A2C implementation presented here. Despite the fact that this difference is likely also an effect of the difference in the amount of iterations per setting, and implementation. The extent of the differences definitely points towards the strength of the critic in ensuring stable learning of the actor.

4.2. Actor Critic

Clearly the A2C performs extremely well compared to the methods used int he previous assignments, even without additional extensions such as AC-Experience Replay (ACER, with PER) or TES-Soft actor Critic (SAC). However, even the simple methods we implement reliably learn the environment in around or less than 50 episodes, when only one agent is playing. Extending this learning rate to A3C, shows that AC methods are able to both efficiently explore and exploit. The mean rewards of the agents are close to the maximum of the environment and are easily above the default reward for the environment (Figure 14) without being specifically tuned to it.

We implement a (somewhat simple) SAC, which, together with the policy anneal and batch size anneal, is responsible for the exceedingly stable exploitation compared to baseline AC, which are typically brittle. This is because we use the three method to tune into a (dynamically scheduled) balance between exploration and exploitation. In the environment they act in they reliably avoid getting stuck in sub-optimal policies for more than some episodes which still return more than half the maximum reward.

4.3. Conclusion

We have found that policy based methods for reinforcement learning are able to perform extremely well on a simple environment such as CartPole, however gradient-based approaches can be extremely unstable. The addition of a critic improves performance immensely relative to the simpler REINFORCE approach. Additionally, a gradient-free approach such as CEM also shows great performance, given a big enough generation size.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Bishop, C. M. and Nasrabadi, N. M. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Christodoulou, P. Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207*, 2019.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. Openai baselines. https://github.com/openai/baselines, 2017.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018a.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018b.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

Xu, Y., Hu, D., Liang, L., McAleer, S., Abbeel, P., and Fox, R. Target entropy annealing for discrete soft actor-critic. *arXiv preprint arXiv:2112.02852*, 2021.

A. Algorithm Blocks

Algorithm 1 REINFORCE (Monte Carlo policy gradient)

Input: Differentiable policy $\pi_\theta(a|s)$ with $\theta \in \mathbb{R}^d$, learning rate η, discount factor γ
Initialization: Randomly initialize θ in $\mathbb{R}^d$.
while *not converged* **do**
  $L(\theta) \leftarrow 0$
  for $m \in 1, .., M$ **do**
    Sample trace $h_0 = \{s_0, a_0, r_0, .., s_{n+1}\}$ [following $\pi_\theta(a|s)$]
    $R \leftarrow 0$
    for $t \in n, .., 1, 0$ **do**
      $R \leftarrow r_t + \gamma \cdot R$; /* backwards through trace */
      $L(\theta) \leftarrow L(\theta) - R \cdot \log\big(\pi_\theta(a|s)\big)$; /* add to total loss */
    end
  end
  $\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\theta)$
end
return $\pi_\theta(a|s)$

Algorithm 2 AC Baseline Subtraction

Input: Policy $\pi_\theta(a|s)$, Value function $V_\phi(s)$, Estimation depth n, Learning rate η.
Initialization: Randomly initialize θ and ϕ.
while *not converged* **do**
  Sample trace $h_0 = \{s_0, a_0, r_0, .., s_{n+1}\}$ under $\pi_\theta(a|s)$
  for $t = 0, .., T$ **do**
    $\hat{Q}_n(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n})$; /* n-step target */
  end
  $\phi \leftarrow \phi - \eta \cdot \sum_t \nabla_\theta \big(\hat{Q}_n(s_t, a_t) - V_\phi(s_t)^2\big)$; /* descent value loss */
  $\theta \leftarrow \theta + \eta \cdot \sum_t \big[\hat{Q}_n(s_t, a_t) \cdot \nabla_\theta \log(\pi_\theta(a|s))\big]$; /* ascent policy gradient */
end
return $\pi_\theta(a|s)$

Algorithm 3 Actor-Critic policy gradient with bootstrapping & baseline subtraction

Input: Policy $\pi_\theta(a|s)$, Value function $V_\phi(s)$, Estimation depth n, Learning rate η.
Initialization: Randomly initialize θ and ϕ.
while *not converged* **do**
  Sample trace $h_0 = \{s_0, a_0, r_0, .., s_{n+1}\}$ [following $\pi_\theta(a|s)$]
  for $t = 0, .., T$ **do**
    $\hat{Q}_n(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n})$; /* n-step target */
    $\hat{A}_n(s_t, a_t) = \hat{Q}_n(s_t, a_t) - V_\phi(s_t)$; /* advantage */
  end
  $\phi \leftarrow \phi - \eta \cdot \sum_t \nabla_\theta \big(\hat{Q}_n(s_t, a_t) - V_\phi(s_t)^2\big)$; /* descent value loss */
  $\theta \leftarrow \theta + \eta \cdot \sum_t \big[\hat{A}_n(s_t, a_t) \cdot \nabla_\theta \log(\pi_\theta(a|s))\big]$; /* ascent policy gradient */
end
return $\pi_\theta(a|s)$

Algorithm 4 Cross-Entropy Method (evolutionary algorithm for policy optimization)

Input: Differentiable policy $\pi_\theta(a|s)$ with $\theta \in \mathbb{R}^d$ & $\theta \sim \mathcal{N}(\mu, \sigma)$, Number of iterations n_{iter}, Number of agents n_{agents}, Elite percentage u.
Initialization: Initialize $\mu_1 \in \mathbb{R}^d$ and $\sigma_1 \in \mathbb{R}^d$.
for $i = 1, .., n_{iter}$ **do**
  Sample parameters $\theta_j \sim \mathcal{N}(\mu_i, \text{diag}(\sigma_i))$ for $j = 1, .., n_{agents}$
  Sample returns $R_j \sim \pi_{\theta_j}(a|s)$
  Select elite set of θ_j given top $u\%$ returns R_j
  μ_{i+1}, $\sigma_{i+1} \leftarrow$ refit Gaussian on elite set
end
return $\pi_\theta(a|s)$

B. A2C Hyperparameter Study

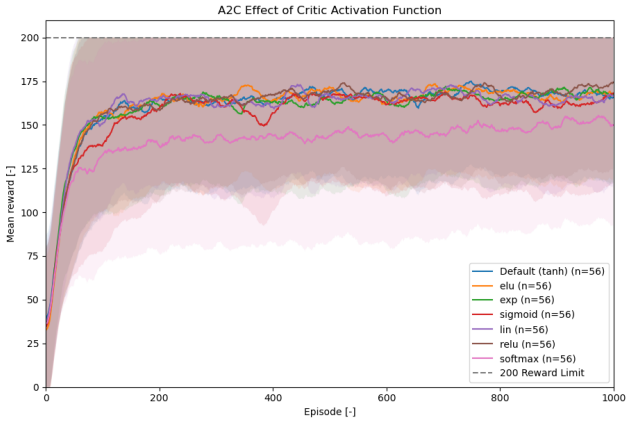

Figure 11. Study of the effect of the critic hidden layer activation functions for A2C.

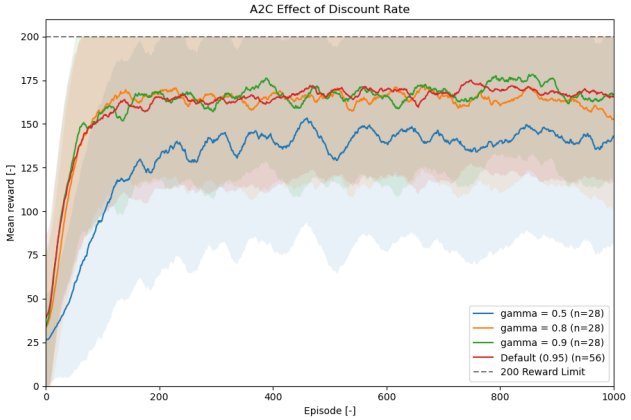

Figure 12. Study of the effect of the discount rate for A2C. A2C seems to be relatively insensitive to the discount rate, as long as it is withing reasonable bounds. In the case of $\gamma = 0.5$, much of the sampled trajectory is discarded, which is not sample efficient.

C. Work Distribution

We again felt that the distribution of work should be highlighted. For this project we split into two groups, one working on the REINFORCE and one working on the A2C algorithms, see Table 3.

Table 3. Work Distribution - Assignment 3 DRL

Algorithm	Responsible
REINFORCE	Rens K., Benjamin S. (equal contribution)
A2C	Lukas W.
CEM	Rens K.

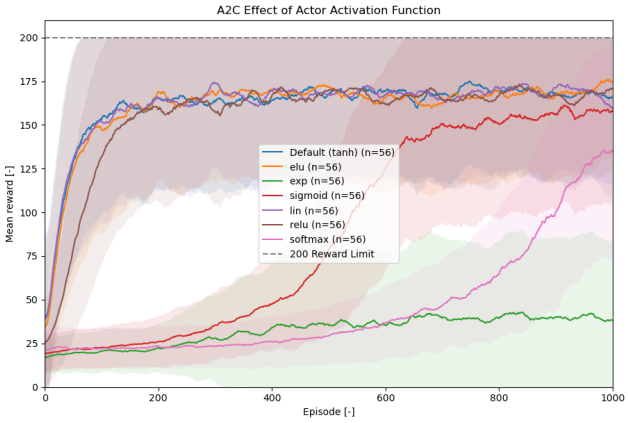

Figure 13. Study of the effect of the actor hidden layer activation functions for A2C. Activation functions which allow for the logits to adequately expressed have a good performance, while functions which squeeze/oversize important parts of the logit domain (exponential, softmax) perform poorly. This is similar for the critic, which is shown in Figure 11 since it is not particularly interesting.

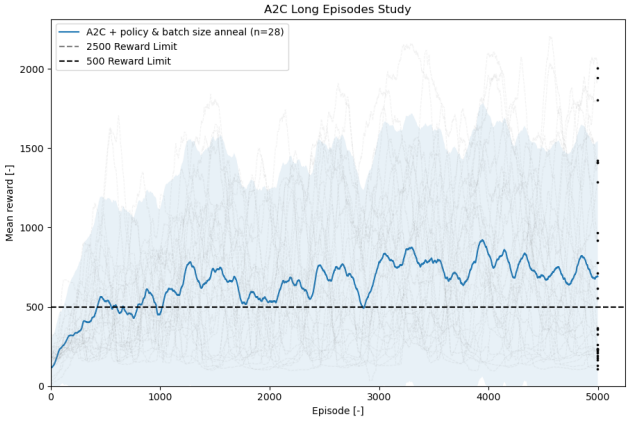

Figure 14. Check whether the restricted environment is acceptable for A2C. This environment has a maximum reward of 2500, which some of the agents approach. The histories in this plot are highly smoothed.