Methods for Deep Reinforcement Learning

Gerben Meijer Ron van Bree Maurice van Leeuwen
Carl Beekhuizen

July 3, 2018

Abstract

State-of-the-art research in RL includes deep learning techniques. This work aims to understand how both traditional and deep RL techniques apply to standard RL problems. The SARSA(λ) , Deep SARSA(λ) and A2C algorithms are implemented and compared using Cart Pole. Deep SARSA(λ) learns to play Snake using CNN, and a feature extraction approach. SAC-X learns sparse rewards tasks using Mountain Cart. Implementation [1] and theory behind these algorithms is explained in-depth, the accuracy of the algorithms is compared, and it is discussed why and how the different approaches perform better on certain RL problems.

Introduction

Reinforcement learning (RL) is widely used in robotics and other applications of AI. While other machine learning techniques are used in these domains, RL specifically aims to

Compared to other machine learning techniques, RL To contrast RL with other machine learning methods, it is often called 'semi-supervised' learning. In states rewards are awarded, which indicates reaching a goal or is a measure of the distance to a goal.

Project Goals

This project is part of a Capita Selecta in Reinforcement Learning for which we define the following goals:

1. Gain understanding and intuition about the theory behind Reinforcement Learning

2. Understand the added value of Deep Reinforcement Learning

[1] The full implementation of algorithms and environments mentioned in this work are publicly available at `https://github.com/Gerryflap/RL_project_common`

Scope

In order to accomplish the goals of this project, a significant amount of research into the field of Reinforcement Learning was done. In this section, we provide a summary of the most important topics covered in this project. In order to understand any RL topic one needs a good understanding of the basics. To cover these basics we studied the RL course by David Silver[1] up to and including lecture 6 (value function approximation) and completed the corresponding exercises. After understanding and implementing the basics we investigated Deep Q-learning[2] and extended it to Deep SARSA(λ) . To cover actor-critic and policy based methods we also studied lecture 7. This was, however, more of an overview and not an in-depth study of the material. Finally, 2 members of the group studied a recent paper by DeepMind where an actor-critic method (SAC-X[3]) for learning in sparse reward environments is proposed. In the upcoming sections these subjects will be summarized and explained.

Contents

1 Background

In order to compare and contrast the various RL methods that are explored within this paper, it is necessary to understand the theoretical underpinnings of the different algorithms implemented. This will serve as a basis for compiling hypotheses on top of which tasks are best suited to which algorithms.

1.1 Reinforcement Learning

Reinforcement Learning concerns itself with devising a policy for picking actions in an environment as to maximize a certain reward. Typically, these environments are modelled as Markov Decision Processes. An MDP is a 5-tuple (S, A, P_a, R_a, γ) where

- S denotes the set of states an environment can be in,
- the set A contains all actions that the agent can take in the environment,
- $P_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of transitioning from state s to s' with action a at time step t,
- R_s^a the reward obtained from taking action a in state s,
- $\gamma \in [0, 1]$ is a reward discount factor controlling the importance of future rewards relative to present rewards.

At each time step t, an agent interacts with the environment by deciding on an action a_t and performing this action in the environment, resulting in a new observation o_{t+1} and reward r_{t+1}. The decision of which action to take is based on the history $H_t = o_1, r_1, a_1, .., a_{t-1}, o_t, r_t$ containing all observations, rewards and actions seen thus far. A distinction is made between environment states s_t^e, which is the environment's private representation fully describing its state, and the agent state s_t^a, which is the agent's internal model of the environment.

In this work, only fully observable environments are considered, meaning that $o_t = s_t^e = s_t^a$. Combining this with the Markov assumption - that the future is independent of the past, given the present - means that the decision procedure for determining an action to take can be optimally decided based on the most recent observation. From this decision procedure, follows a policy $\pi : S \times A \to [0, 1]$, where $\pi(a|s) = p(a_t = a | s_t = s)$. That is, the policy gives the probability of taking action a when in state s. The goal of Reinforcement Learning is to maximize $\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_{t+1} \sim P(\cdot|s_t, a_t),\ a_t \sim \pi(\cdot|s_t),\ s_0 \sim P(s) \right]$, where $P(s)$ denotes an initial environment state distribution and $\gamma \in [0, 1]$, the discount parameter. The total discounted reward from time step t is also referred to as the *return* G_t, giving $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. An agent can make use of a state-value function $v_\pi(s) = \mathbb{E}[G_t | s_t = s]$ giving the expected return from state s when following policy π, or an action-value function $q_\pi(s, a) =$

$\mathbb{E}[G_t|s_t = s, a_t = a]$ giving the expected return starting from state s, taking action a and then following policy π. The true value function is unknown to the agent and should be learned from experience. If both the true value-function and a model of the transition probabilities $P_a(s, s')$ are known, an optimal policy can be derived by maximizing the expected value for a one-step lookahead. From the true action-value function $q(s, a)$ an optimal policy can be derived directly, without the need of transition probabilities. This optimal policy must satisfy the Bellman equation:

$$\begin{aligned} v(s) &= \mathbb{E}[G_t|s_t = s] && (1)\\ &= \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t = s\right] && (2)\\ &= \mathbb{E}[r_{t+1} + \gamma v(s_{t+1})|s_t = s] && (3) \end{aligned}$$

which states that if the agent chooses an optimal action, the value of a state is equal to the immediate reward r_{t+1} plus the expected discounted value of the next state. Dynamic Programming using a fixed policy can be used to explore the state space to observe the MDP rewards and transition probabilities. The Bellman equations can then be used to evaluate whether the derived policy is optimal or not. This is, however, only feasible for small MDPs. Monte-Carlo methods try to solve this by trying to estimate the expected return by taking the mean return of individually sampled episodes. For every state visited, a value function update rule is used to incrementally update the mean:

$$v(s_t) \leftarrow v(s_t) + \frac{1}{N(s_t)}(G_t - v(s_t)) \tag{4}$$

where $N(s_t)$ is the number of times state s_t has been visited so far. By law of large numbers, $v(s)$ converges to the true state-value function. Monte Carlo, however, can only learn from finite episodes sampled from the environment. An alternative is to use Temporal Difference learning which tries to use the current estimation of the value function combined with individual steps observed in episodes to form a better estimate of the value function. To make things more concrete, the simplest Temporal Difference algorithm called TD(0) uses the following update rule:

$$v(s_t) \leftarrow v(s_t) + \alpha(r_{t+1} + \gamma v(s_{t+1}) - v(s_t)) \tag{5}$$

where $r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ is the difference between the current estimate and the estimate incorporating the observed reward. α is a learning parameter that controls the rate at which the value function updates. TD can be generalized to take multiple time steps into account using an n-step return $G_t^{(n)}$.

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n v(s_{t+n}) \tag{6}$$

n-step Temporal Difference learning uses this return in its value function update rule:

$$v(s_t) \leftarrow v(s_t) + \alpha(G_t^{(n)} - v(s_t)) \tag{7}$$

TD(0) uses $n = 1$ and if n is chosen to be infinite this rule is equivalent to Monte Carlo [1]. Higher n use more steps to update the value function, but also introduce more variance in the learning process. No value for n is best for all problems. TD(λ) combines all possible values for n by introducing a weight of $(1-\lambda)\lambda^{n-1}$, with $\lambda \in [0,1]$. the update rule becomes

$$v(s_t) \leftarrow v(s_t) + \alpha(G_t^\lambda - v(s_t)) \tag{8}$$

where $G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \gamma^{n-1} G_t^{(n)}$.

Thus far, episodes of state-action-reward sequences are assumed to be available. Nothing has been said about how the episodes of running the environment are obtained in order to acquire these sequences. A method called Q-learning greedily picks the action that maximizes its current estimate of the action-value function $q(s,a)$, as this does not require a model for $P_a(s,s')$. TD Q-learning therefore uses the following value function update

$$q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma \max_{a'} q(s',a') - q(s,a)) \tag{9}$$

which is calculated when state s transitions to state s' by taking action a giving reward r, and a' is an action performed from state s'. Q-learning does not take into account the action a' that is truly executed, but just assumes it is the most valued action. It therefore completely ignores the policy used to determine actions and is called an off-policy algorithm. SARSA is an on-policy algorithm that uses the truly executed (s,a,r,s',a') quintuplets to update its $q(s,a)$ estimate, giving

$$q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma q(s',a') - q(s,a)) \tag{10}$$

Again, as with TD(λ), this can be generalized to include multiple steps using the n-step q-return $q_t^{(n)}$

$$q_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n q(s_{t+n}, a_{t+n}) \tag{11}$$

to form n-step SARSA

$$q(s,a) \leftarrow q(s,a) + \alpha(q_t^{(n)} - q(s,a)) \tag{12}$$

and subsequently SARSA(λ)

$$q(s,a) \leftarrow q(s,a) + \alpha(q_t^\lambda - q(s,a)) \tag{13}$$

where $q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$.

The way q is updated using (13) is called a forward view formulation. The update is defined in terms of rewards that follow from future steps. Usually this is the most straight-forward manner to express the equations, but it is hard

to implement a forward view since it depends on data that is not yet available. Eligibility traces allow algorithms as SARSA(λ) to be formulated as a backward view. An eligibility trace $\boldsymbol{z}_t \in \mathbb{R}^d$ is a memory that whenever a (s, a) pair is used in an estimated value, the trace for that element is raised by 1. Otherwise the trace decays with a rate determined by the trace-decay parameter $\lambda \in [0, 1]$. Eligibility for a state can be defined recursively:

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s) \end{aligned} \tag{14}$$

Notice that the eligibility trace favours both frequent and recent observations.

Now, to create a backward view formulation of SARSA(λ) the forward view from 11 should be disregarded in favour of eligibility trace E. This leads to the backward view formulation for 13. Here only one next step ($n = 1$) is used, and thus $s_{t+1} = s'$, $a_{t+1} = a'$.

$$\begin{aligned} \delta_t &= r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t) \\ q(s, a) &\leftarrow q(s, a) + \alpha\delta_t E_t(s, a) \end{aligned} \tag{15}$$

Until now it is assumed that q is a discrete function (i.e. a table) that contains entries for (s, a) pairs. But in problems with large MDPs there can be too many states and actions to work with. For example q may contain to many entries to fit in memory, but it also becomes very slow to learn each entry individually. A solution in such problems is to approximate q using a function. This function can be linear or non-linear such as a Neural Networks. Linear functions can be proven to converge in on-policy MC, TD(0) and TD(λ) learning. The approximation of q_π under policy π is $\hat{q}$ uses a vector of weights $\boldsymbol{w}$ that are used in the linear function:

$$\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a) \tag{16}$$

The weights $\boldsymbol{w}$ should be updated in case of error. This can be achieved using stochastic gradient descent (SGD). This leads to updates of $\boldsymbol{w}$ using the gradient $\nabla_{\boldsymbol{w}}$ of the prediction error. Use of SGD allows to sample the gradient, and therefore update $\boldsymbol{w}$ after each step, while the expected update is equal to a full gradient update. Also SGD converges on a global optimum of the error function.

This leads to the following update to $\boldsymbol{w}$ in forward view TD(λ):

$$\Delta\boldsymbol{w} = \alpha(q_t^\lambda - \hat{q}(s_t, a_t, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{q}(s_t, a_t, \boldsymbol{w}) \tag{17}$$

And these are the updates in backward view TD(λ) with eligibility trace E:

$$\begin{aligned} \delta_t &= r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}\boldsymbol{w}) - \hat{q}(s_t, a_t, \boldsymbol{w}) \\ E_t &= \gamma\lambda E_{t-1} + \nabla_{\boldsymbol{w}}\hat{q}(s_t, a_t, \boldsymbol{w}) \\ \Delta\boldsymbol{w} &= \alpha\delta_t E_t \end{aligned} \tag{18}$$

Now it is shown how a function can be learned that approximates q_π, we can extend this to methods that use non-linear functions such as neural networks. Deep Q-Networks (DQN) is a method that uses deep neural networks that approximate q_π.

While linear functions converge in on-policy TD(λ) learning, for non-linear functions this is not true. To use deep neural networks, as is done in Deep Q-Networks (DQN), experience replay and fixed-q parameters are applied to help converge under TD(λ) [2]. Experience replay uses a replay memory $\mathcal{D}$. Here, after taking an action a_t under an ϵ-greedy policy, the quadruplet (s, a, r, s') is stored in the replay memory. This information is not directly for learning. Instead (s, a, r, s') is sampled from $\mathcal{D}$, and this quadruplet is used to determine the prediction error and update q. Another technique that is used are fixed Q-targets. Here two Q-networks are used. A target network with parameters $\boldsymbol{w}^-$ is used to compute the targets. Whereas the Q-network with parameters $\boldsymbol{w}^-$ is the network being updated. Only every c steps the target network parameters $\boldsymbol{w}^-$ are replaced by the Q-network parameters $\boldsymbol{w}$. These two techniques prevent important causes of instability in learning of non-linear functions. These causes are the correlation in the sequence of observations, that small updates to Q can result in significant changes in the policy, and that the target and prediction are correlated. In deep Q-networks the update is defined as the MSE between the current Q-network and the target network:

$$\mathcal{L}_i(w_i) = \mathbb{E}_{(s,a,r,s')\ D_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^- - Q(s, a; w_i)\right)^2\right] \tag{19}$$

1.2 An overview of RL methods

Policy based learning Up until now, the methods covered all discuss the learning of a value function. The policy was always directly derived from the value function in the form of a greedy or ϵ-greedy policy. There are, however, other approaches. Instead of learning the value function, one can also use a learned policy. This method does not use any value function and instead uses the feedback from the environment directly to optimize the policy. Monte-Carlo returns are often used to evaluate the policy based on the feedback from the environment in this case. An advantage of this is that the agent can learn to perform in a continuous action space, something that can not be done using value based learning. A disadvantage is that the Monte Carlo returns have a high variance, which slows down training massively[1].

Actor Critic To solve the high variance caused by using MC for policy learning, a natural extension is to use both a learned value function and a learned policy. These are called the Critic and the Actor. The Actor learns the policy and updates its parameters in a direction suggested by the Critic. The Critic evaluates the policy using policy evaluation methods such as SARSA(λ) . This setup avoids the high variance of using MC in policy methods while keeping all the benefits of policy based learning.

Model-based learning Another important paradigm that has not been covered yet is model-based learning. Up until now, all discussed methods use $Q(s, a)$ to remain model-free. In model-based learning, the agent makes use of an internal model of the environment. This model is learned from interaction with the actual environment. Using this internal model, the agent can simulate ahead and make choices based on what happens. Dyna is a hybrid model-based approach that combines a learned model with policy and/or value function learning. In this course, we choose to focus only on model-free methods.

1.2.1 SAC-X

Some environments have too sparse rewards for the algorithms mentioned above to learn properly. The observed rewards tell so little that learning effectively comes close to random search. A method called Scheduled Auxiliary Control (SAC-X) [3] was proposed to get around this problem. They define a sparse reward problem as finding the optimal policy π^* in an MDP $\mathcal{M}$ with a reward function characterized as

$$R_{\mathcal{M}}(s, a) = \begin{cases} d(s, s_g) \leq \epsilon & = \delta_{s_g}(s) \\ \text{otherwise} & = 0 \end{cases} \tag{20}$$

where $s \in \mathbb{R}^S$, $a \in \mathbb{R}^A$, s_g denotes a goal state, $d(s, s_g)$ is a distance metric between the goal state and state s and $\delta_{s_g}(s)$ defines the reward surface within the ϵ-region. In order to learn from such a sparse reward function a set of auxiliary tasks are used. These are modeled as a set of auxiliary MDPs $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_K\}$ that share the state space S, action space A and transition probabilities $P_a(s, s')$ with the main task $\mathcal{M}$, but have separate auxiliary reward functions $R_{\mathcal{A}_1}(s, a), ..., R_{\mathcal{A}_K}(s, a)$. With these reward functions the following task specific return is defined as

$$G_{\mathcal{T}}(\tau_{0:\infty}) = \sum_{t=0}^{\infty} \gamma^t R_{\mathcal{T}}(s_t, a_t) \tag{21}$$

where $\mathcal{T} \in T = \mathcal{A} \cup \{\mathcal{M}\}$. Using $G_{\mathcal{T}}$ the task specific action value function $Q_{\mathcal{T}}(s_t, a_t)$ is defined as follows:

$$Q_{\mathcal{T}}(s_t, a_t) = R_{\mathcal{T}}(s_t, a_t) + \gamma \mathbb{E}_{\pi_{\mathcal{T}}} \left[G_{\mathcal{T}}(\tau_{t+1:\infty}) \right] \tag{22}$$

where $\pi_{\mathcal{T}} = \pi_\theta(a|x, \mathcal{T})$ is the intention specific policy characterized by parameters θ. Every intention is optimized using loss function $\mathcal{L}$ to select optimal actions for it tasks starting from an initial state drawn from a state distribution $p(s|\mathcal{B})$ obtained by following any other policy $\pi(a|s, \mathcal{B})$ with $\mathcal{B} \in T = \mathcal{A} \cup \{\mathcal{M}\}$. That is,

$$\mathcal{L}(\theta) = \mathcal{L}(\theta; \mathcal{M}) + \sum_{k=1}^{|\mathcal{A}|} \mathcal{L}(\theta; \mathcal{A}_k) \tag{23}$$

with

$$\mathcal{L}(\theta; \mathcal{T}) = \sum_{\mathcal{B} \in T} \mathbb{E}_{p(s|\mathcal{B})} \left[Q_{\mathcal{T}}(s, a) \mid a \sim \pi_\theta(\cdot|s, \mathcal{T}) \right] \tag{24}$$

Sampling from a state distribution obtained from following other auxiliary tasks $\mathcal{B}$ is done to create compatible policies for each intention. The $Q_{\mathcal{T}}(s, a)$ values are obtained using a parameterized predictor $\hat{Q}^\pi_{\mathcal{T}}(s, a; \phi)$. Using this estimator, together with a replay buffer B containing gathered trajectories τ, a gradient based approach can be taken in optimizing policy parameters θ, giving

$$\nabla_\theta \mathcal{L}(\theta) \approx \sum_{\substack{\mathcal{T} \in T \\ \tau \sim B}} \nabla_\theta \mathbb{E}_{\substack{\pi_\theta(\cdot|s_t, \mathcal{T}) \\ s_t \in \tau}} \left[\hat{Q}^\pi_{\mathcal{T}}(s_t, a; \phi) + \alpha \log \pi_\theta(a|s_t, \mathcal{T}) \right] \tag{25}$$

where $\mathbb{E}_{\pi_\theta(\cdot|s_t, \mathcal{T})} \left[\alpha \log \pi_\theta(a|s_t, \mathcal{T}) \right]$ corresponds to entropy regularization with weighting parameter α. The $Q(s, a)$ estimator $\hat{Q}^\pi_{\mathcal{T}}(s, a; \phi)$ is optimized using Retrace [4] in the following loss:

$$\begin{aligned}
\min_\phi L(\phi) &= \mathbb{E}_{(\tau, b, \mathcal{B}) \sim B} \left[(\hat{Q}^\pi_{\mathcal{T}}(s, a; \phi) - Q^{\text{ret}})^2 \right] \\
Q^{\text{ret}} &= \sum_{j=i}^{\infty} \left(\gamma^{j-i} \prod_{k=i}^{j} c_k \right) \left[R_{\mathcal{T}}(s_j, a_j) + \delta_Q(s_i, s_j) \right] \\
\delta_Q(s_i, s_j) &= \mathbb{E}_{\pi_{\theta'}(a|s, \mathcal{T})} \left[Q^\pi_{\mathcal{T}}(s_i, \cdot; \phi') \right] - Q^\pi_{\mathcal{T}}(s_j, a_j; \phi') \\
c_k &= \min \left(1, \frac{\pi_{\theta'}(a_k|s_k, \mathcal{T})}{b(a_k|s_k, \mathcal{B})} \right)
\end{aligned} \tag{26}$$

where trajectory τ is sampled from the replay buffer, b is the policy under which actions in τ were generated and $\mathcal{B}$ is the intention during action sampling. θ' and ϕ' denote parameters of the policy and Q-estimator (both neural networks) which are periodically switched with the current parameters θ and ϕ to improve learning stability.

A scheduler $\mathcal{S}$ is used to determine what the current intention of the agent is, based on previous intentions. Let ξ denote the period in which the scheduler can switch between tasks, and H denote the total number of task switches in an episode. The H scheduling choices are denoted by $\mathcal{T}_{0:H-1} = \{\mathcal{T}_0, ..., \mathcal{T}_{H-1}\}$. The return with respect to the main task given the scheduling choices $\mathcal{T}_{0:H-1}$ can then be defined as

$$G_{\mathcal{M}}(\mathcal{T}_{0:H-1}) = \sum_{h=0}^{H-1} \sum_{t=h\xi}^{(h+1)\xi-1} \gamma^t R_{\mathcal{M}}(s_t, a_t) \tag{27}$$

where $a_t \sim \pi_\theta(\cdot|s_t, \mathcal{T}_h)$. The scheduling policy from which intentions are sampled is denoted by $P_{\mathcal{S}}(\mathcal{T}|\mathcal{T}_{0:h-1})$. A second goal of optimizing objective $\mathcal{L}(\mathcal{S})$ for learning scheduler $\mathcal{S}$ is defined as

$$\mathcal{L}(\mathcal{S}) = \mathbb{E}_{P_{\mathcal{S}}} \left[G_{\mathcal{M}}(\mathcal{T}_{0:H-1}) \mid \mathcal{T}_h \sim P_{\mathcal{S}}(\mathcal{T}|\mathcal{T}_{0:h-1}) \right] \tag{28}$$

The individual intention policies are fixed here and not optimized with respect to θ. The schedule returns are approximated using $Q(\mathcal{T}_{0:h-1}, \mathcal{T}_h) \approx \mathbb{E}_{P_\mathcal{S}}\left[G^\tau_\mathcal{M}(\mathcal{T}_{h:H}) \mid \mathcal{T}_{0:h-1}\right]$. As the total number of tasks is small, $Q(\mathcal{T}_{0:h-1}, \mathcal{T}_h)$ can be stored in a table. Monte Carlo estimates are used to approximate this value:

$$Q(\mathcal{T}_{0:h-1}, \mathcal{T}_h) = \frac{1}{M} \sum_{i=1}^{M} G^\tau_\mathcal{M}(\mathcal{T}_{h:H}) \tag{29}$$

where M is the number of runs the estimate is based on and $G^\tau_\mathcal{M}$ is the return w.r.t. task $\mathcal{T}$ when following trajectory τ. Using the approximated return values, tasks obtained from the scheduler are sampled from a Boltzmann distribution with temperature parameter μ.

$$P_\mathcal{S}(\mathcal{T}|\mathcal{T}_{0:h-1}; \mu) = \frac{\exp(\mathbb{E}_{P_\mathcal{S}}[G_\mathcal{M}(\mathcal{T}_{h:H})]/\mu)}{\sum_{\hat{\mathcal{T}}_{h:H}} \exp(\mathbb{E}_{P_\mathcal{S}}[G_\mathcal{M}(\hat{\mathcal{T}}_{h:H})]/\mu)} \tag{30}$$

2 Methodology

In this research a number of different RL algorithms are applied to multiple environments in order to acquire more intuition about these methods and their differences. The pairing of algorithms to environments is made such that anticipated helpful or problematic characteristics are exposed in the results. The RL algorithms that are implemented and compared in this research are SARSA(λ) , Deep SARSA(λ) , Actor Critic methods. Deep SARSA(λ) is applied both using a fully connected neural network and a convolutional neural network (CNN). These algorithms are core RL methods, based on value learning, and are extended with policy learning in A2C. Model based learning (for example Dyna-Q [5]) is left out of the comparison.

Explain how the graphs work

2.1 Environments

Various environments have been used to evaluate the algorithms. This section provides a detailed description of all environments.

2.1.1 Cart Pole

Cart Pole[2] is an environment wherein the agent is required to balance an inverted pendulum by moving the cart on which the pole is balanced.

- **State Space:** The state space is comprised of 4 continuous valued parameters, namely the position of the cart, the angle of the pole and their first temporal derivatives.

[2]OpenAi's Cart Pole can be found at: `https://gym.openai.com/envs/CartPole-v1/`

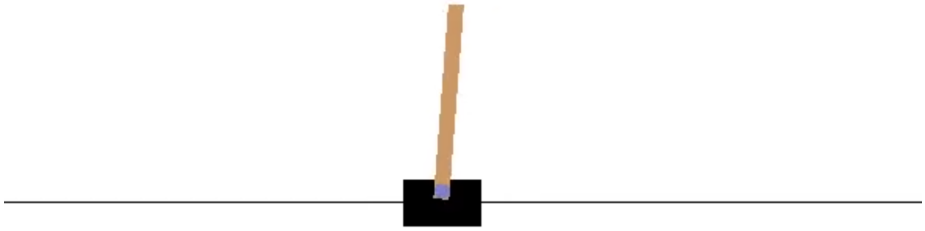

Figure 1: OpenAi's CartPole environment [6]

- **Action Space:** AT every time step, the agent has the choice of applying an acceleration to the craft in either the left or right direction.
- **Reward:** A positive (+1) reward is received for every time-step where the pole remains in the range $-12°$ to $12°$. The environment terminates and returns 0 reward once this range has been exceeded.

2.1.2 Snake

In Snake[3] it is a goal to control a Snake and let it grow as long as possible. The Snake grows from eating fruits. Every time step in the game the head of the Snake moves one block ahead, in one of four directions. The body and tail of the snake follows the trajectory the head of the snake already passed. As the snake grows, it becomes harder and harder to let the snake not bump into itself. When that happens the game ends.

- **State Space:** The state space of snake is offered as two distinct variants, namely:
  - **Feature Extracted:** This state space returns the distance to collision in each of the directions $\{-\frac{3}{4}\pi, -\frac{1}{2}\pi, -\frac{1}{4}\pi, 0, \frac{1}{4}\pi, \frac{1}{2}\pi, \frac{3}{4}\pi\}$ relative

[3]The implementation of Snake we used is: `https://github.com/av80r/Gym-Snake`

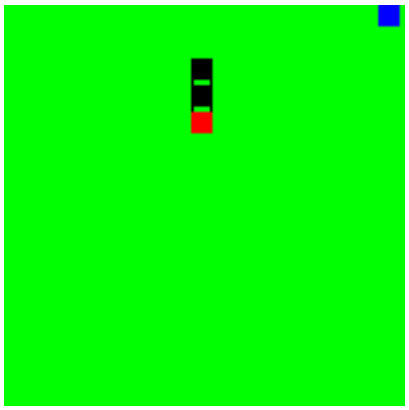

Figure 2: The 15x15 snake environment, with a red snake head and blue food. [7]

to the snake's head. Furthermore, the agent receives the relative argument and distance of the food with respect to the snake's head.

- **Visual:** In Visual mode, the environment returns a 3-channel, boolean, 8x8 grid of the 'pixels' of the snake. The channels are comprised of the {Snake Body, Snake Head, Food}. The environment rotates the state space such that the snake is always moving downwards. This decreases the size of the state-space by eliminating rotational symmetries.

- **Action Space:** At every step, the snake has the choice of turning right, left or continuing straight.

- **Reward:** A $+1$ reward is returned whenever the snake eats a fruit. A -1 reward is returned when the snake dies.

2.1.3 Mountaincar

The Mountaincar environment used is a modified version of the Mountaincar[4] environment by OpenAI. In Mountaincar the goal is to get a small car to a flag that is situated on a hill. The only problem is that this hill quite steep and the car cannot just climb it without any momentum. Therefore the agent should learn to build momentum in order to climb the hill and reach the flag. This problem is very hard since the agent effectively has to perform a random search to find the goal.

[4]The OpenAI Mountaincar environment can be found here: `https://github.com/openai/gym/wiki/MountainCar-v0`

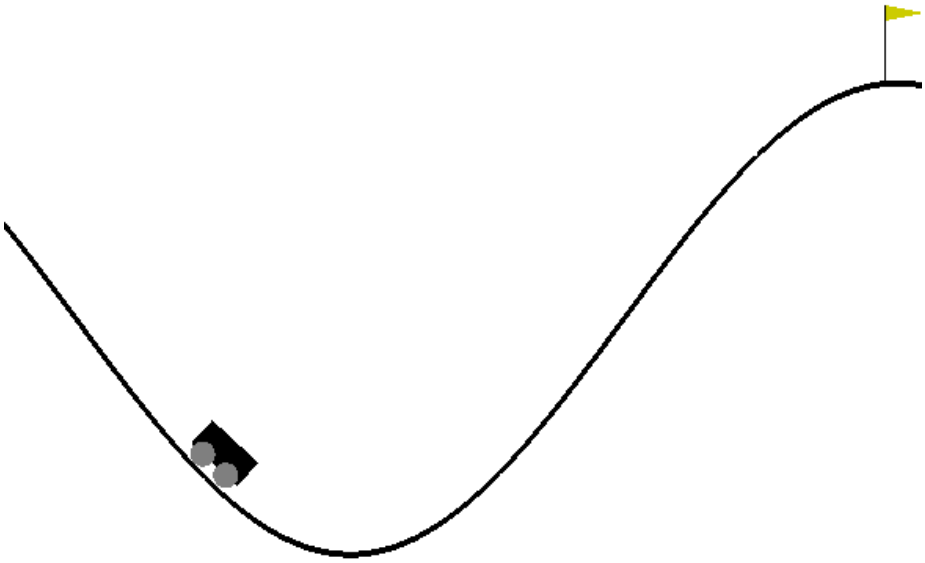

Figure 3: A frame from the OpenAI Mountaincar environment

- **State Space:** The state space of Mountaincar consists of only two continuous values, position and velocity.
- **Action Space:** The agent can accelerate left, accelerate right or do nothing.
- **Reward:** We modified the reward to give +1 reward if a terminal state (the flag) is reached before a 1000 steps. At 1000 steps the environment is terminated and no reward is given.

2.2 SARSA(λ) vs Deep SARSA(λ)

In the first comparison SARSA(λ) and Deep SARSA(λ) will be compared on the Cart Pole environment. Both methods will be applied with a range of λ values to investigate the effect of this parameters. Furthermore, the effect of noise in the state space will be explored by applying normally distributed noise to the state vector, scaled respective to the range of every feature in the vector.

2.2.1 Experiment setup

For every environment state observation $\mathbf{s} \in S \subset \mathbb{R}^N$ scaled Gaussian noise vector $\mathbf{z} \in \mathbb{R}^N$ is added element-wise, where N differs for each environment. For every feature s_i in $\mathbf{s}$, $z_i = x_i y_i$, with x_i being the scaling factor depending

on the range of the feature and $y_i \sim \mathcal{N}(0, \sigma)$. The Cart Pole state space is a strict subset of $\mathbb{R}^4$ and its scaling factors are as follows:

- $x_0 = 2.4$, corresponding to the cart's position
- $x_1 = 3.6$, corresponding to cart velocity
- $x_2 = 0.26$, corresponding to the pole angle in radians
- $x_3 = 3.5$, corresponding to the pole's velocity at the tip

2.2.2 SARSA(λ) implementation and configuration

The table-based SARSA(λ) method is applied to a Cart Pole environment. In table-based methods the learned $Q(s, a)$ function is a table, and therefore s and a must be discrete values. The Cart Pole environment has a continuous state space which must be discretized in order to apply discrete learning methods. This results in an unavoidable loss of information about the state space. A Q-table must contain enough entries so that the algorithm can accurately discern between the different rewards it gets in (slightly) different states. Using a Q-table that has more entries, but with a same amount of learning, results effectively in less learning per entry on average. This makes that a Q-table that is too large may become too hard to learn.

The SARSA(λ) implementation uses epsilon decay that decays after every step. And the implementation uses an eligibility trace to apply backward-view updates to Q. The eligibility trace is updated every step and reset to 0 every episode.

The applied discretization is as follows. Since the x_0 and x_2 are bounded, their value can be normalized. Before rounding the value is multiplied by a constant, this determines how many discrete values exist for that variable. The velocities x_1 and x_3 are unbounded. To prevent the discretized state space from becoming too large, the values are bounded by applying a square root. Another effect is that discretization is more precise at low velocities (in both directions) than at high velocities. This improves precision when the cart balances by moving left and right, i.e. when there are many zero crossings in the velocity.

$$x'_0 = \lfloor 5 \cdot \frac{2.4 + x_0}{4.8} + 0.5 \rfloor$$

$$x'_1 = \lfloor \text{sign}(x_1) \ \left(5 \cdot \sqrt{|x_1|} + 0.5 \right) \rfloor$$

$$x'_2 = \lfloor 10 \cdot \frac{x_2 + 0.26}{0.52} + 0.5 \rfloor$$

$$x'_3 = \lfloor \text{sign}(x_3) \left(7 \cdot \sqrt{|x_3|} + 0.5 \right) \rfloor$$

Configuration SARSA(λ) is used together with ϵ decay. The parameters for this are $\epsilon_{start} = 0.7$, $\epsilon_{min} = 0.0$, and $\epsilon_{decay} = 0.9998$. A slower decay ensures the agent learns enough before acting greedily. The parameters for decay are determined experimentally, making sure that the agent explores enough to reach a good accuracy, but also that epsilon decreases quick enough so that it can be observed how quick the agent can learn. After 10,000 steps of learning (which translates to 100 episodes of 100 steps) $\epsilon = 0.14$, so there is still significant exploration. At 30,000 steps $\epsilon = 0.002$ and exploration practically ends. In the experiment $\gamma = 1.0$.

The experiment aims to show the impact of measurement noise at different λ. The experiment that is setup trains the SARSA(λ) algorithm in the Cart Pole environment. The applied noise levels are $\sigma = [0.0, 10^{-2}, 10^{-1}, 10^{0}]$. These noise levels represent situations where there is no noise at all, where noise is equal in amplitude as the original signal, and two situation with slight and substantial noise levels. The experiments are performed with lambda values $\lambda = [0, 0.5, 0.75, 0.9, 1.0]$. This shows the effects of only updating the current state $\lambda = 0$, updating the full trajectory $\lambda = 1$, and a few intermediate situations. Early experimentation showed that lambdas closer to 1 exhibit the most interesting behavior w.r.t learning in a noisy environment.

2.2.3 Deep SARSA(λ) implementation and configuration

Implementation For this comparison, an implementation of SARSA(λ) with Neural Networks as function approximators was also required. As noted in the background, simply using the online approach used for normal SARSA(λ) is not guaranteed to converge and will not perform well. As a solution replay memory and parameter fixing were proposed in the DQN paper by DeepMind[2]. Both of these methods were implemented to improve stability and sample efficiency. To calculate the returns a forward-view implementation of SARSA(λ) was used. This implementation requires trajectories and cannot function with a single SARSA tuple. Therefore the replay memory is a list of trajectories. Every training step 32 trajectories are sampled from the replay memory. For each of these trajectories a random timestep in the trajectory is chosen for this s, a tuple the return is calculated and added to the batch of return values. Gradient descend is then applied such that the network will reduce the error between these new Q returns and its outputs for each of the s, a tuples in the batch.

An optimization implemented to speed up computations is the capping of lambda values. The λ_{min} parameter can be used to assign a minimal value for λ^k. Values for $TD(n)$ will then not be calculated beyond $\lambda_{min} > \lambda^k$. The distribution is then renormalized such that the total probability distribution adds up to 1. This can massively shorten the number of states that need to be forward passed through the Neural Network. For Cart Pole this can mean more than a 10 times speed increase for trajectories of length 500 and $\lambda \leq 0.9$ while having no significant effect on the outcome.

Another choice made in the implementation is the rescaling of the λ distri-

Parameter	Value
γ	1.0
λ_{min}	10^{-3}
reward scale (is multiplied with every reward to scale rewards)	0.01
fixed network synchronization frequency (in training steps)	100
Replay memory size (in trajectories)	1000
Learning rate	0.001
Optimizer	Adam[8]

Table 1: Hyperparameters used for Deep SARSA(λ)

bution. The formula for forward view $TD(\lambda)$ is defined as

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

This works well for an infinite trajectory, but the weights do not sum to one when the trajectory is not infinite. Since in practice this is never the case, the choice was made to rescale the weights differently. Instead of multiplying with $(1-\lambda)$, all weights are multiplied with $\frac{1}{\sum_{n=1}^{T} \lambda^{n-1}}$ to rescale the total sum of weights to one.

While this approach does fix the issue, it does not agree with the theory that $TD(\lambda = 1)$ equals Monte Carlo. Instead it is the mean of all $TD(n)$ returns. At a late stage in the project, we have found that there was a proposed solution in the slides that we overlooked during implementation. This resolves the issue by adding the remaining probability mass to the MC return. This implementation difference only becomes significant with λ values close to 1 where there is some probability mass left at the end of the trajectory. Because of time constraints this fix was only applied to some λ values. This will be explained in the results section of the report.

Configuration As applying a parameter sweep on all hyperparameters would go way beyond the scope of this project, most of the parameters have been derived by manually tweaking the configuration until it was performing as expected. Some of the hyperparameters used for Deep SARSA(λ) on Cart Pole can be found in Table 1. The neural network used consists of 3 fully-connected layers. The first and second hidden layers have 150 and 50 output units respectively and use the $ReLU$ activation function. The output layer has 2 outputs, one for each action in the action space. The output layer uses no activation function (linear). The hyperparameters ϵ_{start}, ϵ_{min} and ϵ_{decay} have been determined by a parameter sweep that was done for another experiment, which uses similar hyperparameters and compares Deep SARSA(λ) to an actor critic approach. The values for these parameters are $\epsilon_{start} = 1.0$, $\epsilon_{min} = 0.0$ and $\epsilon_{decay} = 0.9995$, where decay is applied every step.

2.3 Convolutional vs Transformed

This experiment serves to explore and evaluate the effects of using raw pixel data with a convolutional deep network in lieu of feature extracted parameters with Deep SARSA(λ) . The reason for performing this experiment is to explore the enhancements and trade-offs of convolution across all of the pixel data in comparison to a limited subset of the same information extracted as features.

For this purpose, the snake environment is used. The snake environment is pertinent as the length of the snake is dynamic and thus difficult to encapsulate as the statically sized input array required as input to Deep SARSA(λ) without resorting to using an input of the size of the maximal snake length. The snake environment offers either boolean pixel values over which convolution can be applied or a feature extracted version which encapsulates far less information.

Our hypothesis is that the feature extracted version of snake would train far more rapidly than that of the visual variant, yet would asymptotically approach a less-optimal strategy for controlling the snake. The intuition as to why we propose this hypothesis is that the feature-extracted variant has a much smaller state-space and thus the time required to explore it to the degree that the function approximator can reflect a useful value-function will be shorter. The fact that the feature extracted version does not have access to the position of all of the body sections of the snake should ultimately limit its performance as there are structures that it is incapable of representing. An example of such a structure is when the snake encircles itself in a manner wherein it is impossible for itself to escape without dieing.

2.4 Value based vs Actor Critic

In this experiment the value function learning approach introduced earlier, Deep SARSA(λ) , will be compared to an actor critic method. This is done to gain more intuition about the differences between the methods. Support for a continuous action space, which is a major advantage for AC methods, will not be used here. Instead the Actor model will have softmaxed output that models the policy of the agent. The actor critic method will use the advantage function $A(s, a) = Q(s, a) - V(s)$ and is therefore referred to as A2C (Advantage Actor Critic). Both Deep SARSA(λ) and A2C will use the same configuration for the value function approximator, but Deep SARSA(λ) will use an ϵ-greedy policy whereas A2C uses another neural network to model the policy distribution for a given state.

2.4.1 Policy Learning

The A2C policy is optimized using the policy optimization inspired by the A3C (Asynchronous Advantage Actor Critic)[9] algorithm. More formally, we maximize the following function:

$$L(\theta) = \mathbb{E}_{\pi(\cdot|s,\theta)}[A(s, a) - \alpha \log(\pi(a|s, \theta))]$$

Where $\pi(\cdot|s, \theta)$ denotes the probability distribution of the policy network over the action space for state s under the "live" parameters θ and $A(s, a)$ denotes the advantage function. α is the entropy regularization factor used to balance exploration versus exploitation.

To calculate this loss, the advantage function $A(s, a)$ is required. Since there is no way to directly infer the value of $V(s)$ from the value network, it is instead computed using $\mathbb{E}_{\pi(\cdot|s,\theta')}Q(s, a; \phi')$. Note that the fixed network parameters θ' and ϕ' are used here to increase stability.

2.4.2 Model configuration

To acquire good parameters for both policies, a rough parameter sweep has been performed for both policies. For the ϵ-greedy policy, multiple configurations of ϵ decay and initial and minimum ϵ values have been considered. For the trained policy network only the entropy regularization term α has been explored.

Parameter Sweep Results For Deep SARSA(λ) the parameter sweep performed well for two types of configurations. Configurations where epsilon was constantly zero and configurations where epsilon was quickly (within 100 episodes) decayed to a negligible amount. Although runs with $\epsilon = 0$ converged quickly, reaching max score in roughly 10 episodes, they were also unstable after convergence. The other type of configuration, where epsilon was decayed within roughly 100 episodes, explored more in the beginning and was therefore more stable in the end. Because of this result, $\epsilon_{start} = 1.0$, $\epsilon_{min} = 0.0$ and $\epsilon_{decay} = 0.9995$ were chosen as the final policy parameters. It is important to note here that ϵ_{decay} is multiplied with ϵ at every training step, not every episode. $\epsilon_{decay} = 0.9995$ means that after 100 episodes of on average 100 steps (meaning 10,000 steps in total), we have $\epsilon = 0.006729527$.

A2C proved to be very sensitive to changes to the entropy regularization term α. The parameter sweep showed an optimal value of $\alpha = 0.01$.

Neural Network Hyperparameters All networks used here have the same layout and learning rate. The layout is equal to the layout used for Deep SARSA(λ) in the earlier experiment. For the Actor network the output activation has been switched to softmax instead of linear. This means that the actor critic agent has 2 similar, but separate, neural networks.

2.5 SAC-X

In order to evaluate the added value of SAC-X over normal Actor Critic methods, SAC-Q will be tested on a sparse reward task with and without auxiliary rewards. SAC-Q without auxiliary tasks is essentially a normal Actor Critic algorithm. In order to show the advantage that auxiliary tasks provide, the algorithm with and without these tasks will be applied to the Mountaincar environment introduced earlier. In the upcoming sections the implementation

of SAC-Q, the configuration of the hyperparameters and the defined auxiliary tasks will be discussed.

2.5.1 Implementation

In order to implement SAC-Q, multiple large design decisions have been made. The SAC-X paper was unclear to us at times, which led to some implementation details that are not completely similar to the paper. Since there was no reference implementation available everything is based on the paper, papers cited by the paper and our own knowledge.

Updating the value network In SAC-X, equation 13 is proposed to optimize the value functions. This equation left us with a couple of questions. First off, it specifies the $L(\phi)$ over the whole replay, while updates on neural networks are usually done in randomly sampled batches. Secondly, from the second line onwards, there is a value i that is used but never defined. This value indicates the index of the timestep in the trajectory from which the Q return is calculated. The paper does not specify how this is chosen. Both of these issues are solved by generating random batches. 32 trajectories are randomly sampled from the replay memory and then a random i is chosen in that trajectory for which the return is calculated.

The third issue is that δ_Q is defined differently in the original retrace paper[4]. In SAC-X it is $\mathbb{E}_\pi[Q(s_i, \cdot)] - Q(s_j, a_j)$, whereas the retrace paper defines it as $\mathbb{E}_\pi[Q(s_j + 1, \cdot)] - Q(s_j, a_j)$. Overall, the definition in the original retrace paper makes more sense in our opinion and therefore the implementation is based off of their equations.

Updating the policy network The paper states the following gradients for the policy network:

$$\nabla_\theta \mathcal{L}(\theta) \approx \sum_{\substack{\mathcal{T} \in T \\ \tau \sim B}} \nabla_\theta \mathbb{E}_{\substack{\pi_\theta(\cdot|s_t, \mathcal{T}) \\ s_t \in \tau}} \left[\hat{Q}^\pi_\mathcal{T}(s_t, a; \phi) + \alpha \log \pi_\theta(a|s_t, \mathcal{T}) \right] \tag{31}$$

The issue here is that, according to our understanding, the entropy regularization is supposed to increase entropy. Shannon entropy is defined as $-\sum p \log(p)$ and is always larger than or equal to 0. But then entropy regularization defined here is effectively the same equation without the minus. Therefore maximizing the equation above would effectively minimize the entropy. In practice, this caused the agent to rapidly converge to a suboptimal policy in early experiments that we ran. This is under the assumption that α is positive. Since using a negative α was regarded as a bad solution, we opted for the following new equation for calculating gradients:

$$\nabla_\theta \mathcal{L}(\theta) \approx \sum_{\substack{\mathcal{T} \in T \\ \tau \sim B}} \nabla_\theta \mathbb{E}_{\substack{\pi_\theta(\cdot|s_t, \mathcal{T}) \\ s_t \in \tau}} \left[\hat{Q}^\pi_\mathcal{T}(s_t, a; \phi) - \alpha \log \pi_\theta(a|s_t, \mathcal{T}) \right] \tag{32}$$

In this equation, the entropy regularization has been multiplied with -1 to fix the issue presented above.

SAC-Q Scheduler For SAC-Q, the idea is to use a scheduler which is also trained. The way of updating this scheduler and how the policy is determined was confusing to us. In equation 12 in the paper (Eq. 29 in our report) the way of calculating Monte Carlo returns is listed, but in algorithm 3 from the SAC-X paper a different approach is defined. Here, M is continuously increased as a task is visited more often. This causes the size of the gradients to decrease massively, rendering them useless by the time that any task has actually been learned. To solve this issue in the time constraints, we have opted to replace the $\frac{1}{M}$ by a constant learning rate instead. This is not completely true to the paper, but does work in practice and solves the issues with the solution in algorithm 3 of the paper.

Parallelism An attempt was made to implement the full asynchronous SAC-X system as proposed in the 3 pseudo-code algorithms given by the paper. This attempt was not successful since debugging such a complicated system proved to be almost impossible in the given time. Therefore we decided to implement the algorithm in a sequential fashion. The actor generates a trajectory and then the learner trains on a fixed number of batches. This proved to be way simpler to implement and verify. This method also does not necessarily conflict with the SAC-X paper, since the paper states that the parallelism was only used to speed up experimentation.

2.5.2 Configuration

A large part of the configuration for SAC-X has been established by informal experimentation. There is not guarantee that these are optimal hyperparameters, but both task configurations do use the same hyperparameters in order to make a fair comparison. The important hyperparameters used in the experiment are listed in Table 2.

Both the value and the policy network use a configuration similar to SAC-X. We use one shared layer with 100 output units and $ReLU$. This shared layer is not shared between the value and policy network, but is shared between intentions in both networks respectively. This is similar to the configuration in SAC-X although they use more than one layer. Both networks also have separate subnetworks for each intention. All of these have another fully connected layer with 100 output units and $ReLU$. The final layer for all of these networks is the output layer with three output units corresponding to each action. The output layer uses a linear activation function for the value network and a softmax activation for the policy network. Note that this architecture is different than the architecture used in the SAC-X paper. In the paper the value network has only one output neuron and is fed the action as an input value. We chose a different implementation because it allowed for a simpler implementation, but our method is only possible for discrete action spaces.

Parameter	Value
λ	0.95
γ	0.993
λ_{min}	10^{-2}
reward scale (is multiplied with every reward to scale rewards)	0.03
fixed network synchronization frequency (in training steps)	100
Replay memory size (in trajectories)	3000
Learning rate (Value network)	0.001
Learning rate (Policy network)	0.0001
α (entropy regularization)	0.005
Learning steps per episode	30
η (Boltzmann distribution temperature)	0.1
ξ (Scheduler period)	200
Optimizer	Adam[8]

Table 2: Hyperparameters used for SAC-X

2.5.3 Tasks

The SAC-X algorithm relies on predefined auxiliary tasks with their own rewards in order to improve over other algorithms. These auxiliary tasks require separate rewards defined on the same MDP. The first two auxiliary tasks are `GO_LEFT` and `GO_RIGHT`. These are very simple tasks that reward the agent for either picking the left or right action. The formal definitions of the rewards are given here:

$$R_{\texttt{GO\_LEFT}}(\boldsymbol{s}, \boldsymbol{a}) = \begin{cases} 1 & \text{iff } \boldsymbol{a} = \texttt{ACTION\_LEFT} \\ 0 & \text{else} \end{cases}$$

$$R_{\texttt{GO\_RIGHT}}(\boldsymbol{s}, \boldsymbol{a}) = \begin{cases} 1 & \text{iff } \boldsymbol{a} = \texttt{ACTION\_RIGHT} \\ 0 & \text{else} \end{cases}$$

The third auxiliary task rewards the agent for reaching a given speed. The idea is to teach the agent how to keep its speed up or how to gain speed if it is going slow. The reward function is given below, where $\boldsymbol{s}_1$ is the velocity of the cart:

$$R_{\texttt{GO\_FAST}}(\boldsymbol{s}, \boldsymbol{a}) = \begin{cases} 1 & \text{iff } \boldsymbol{s}_1 \geq 0.03 \\ 0 & \text{else} \end{cases}$$

3 Results and Discussion

In this section the results of the experiments introduced in the methodology will be shown and discussed.

3.1 SARSA(λ) vs Deep SARSA(λ)

3.1.1 SARSA(λ)

The results for SARSA(λ) are listed and discussed in this section.

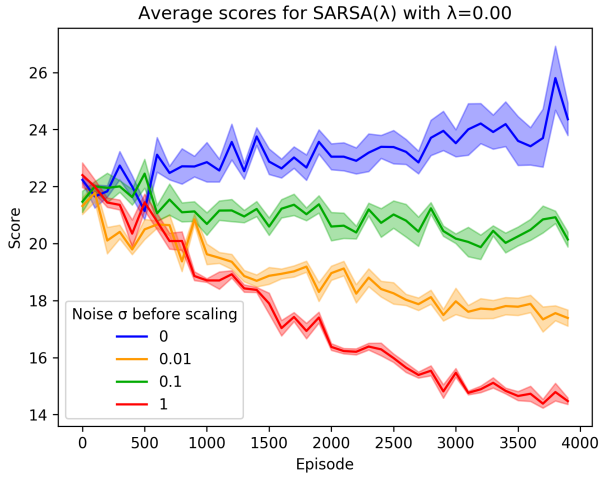

Figure 4: 100 episode average score averaged over 5 runs for SARSA(λ) with $\lambda = 0$ on Cart Pole

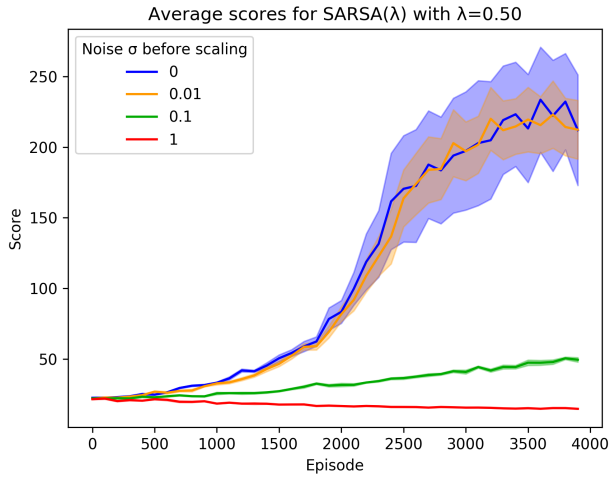

Figure 5: 100 episode average score averaged over 5 runs for SARSA(λ) with $\lambda = 0.50$ on Cart Pole

The first two plots, Figure 4 and Figure 5, show the performance of SARSA(λ) using $\lambda = 0$ and $\lambda = 0.5$ respectively. In neither of the cases SARSA(λ) was able to acquire the highscore of 500 points. Our hypothesis is that this happens because the next states are often the same as the current state due to the discretization. Therefore, low values of λ calculate the return only based off of Q values for the same state (and of course the rewards). This appears to massively impair the learning ability of the algorithm.

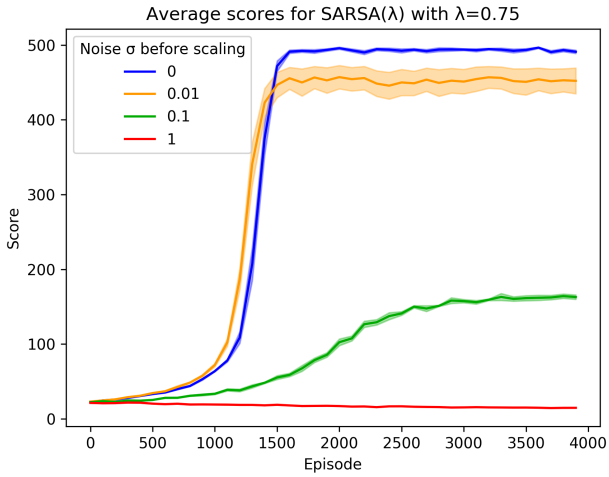

Figure 6: 100 episode average score averaged over 5 runs for SARSA(λ) with $\lambda = 0.75$ on Cart Pole

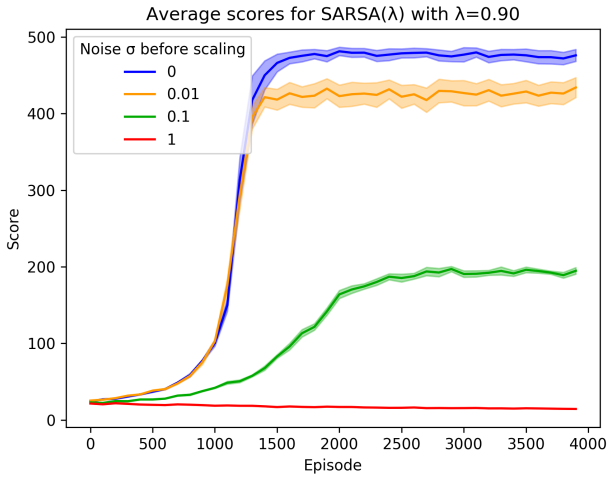

Figure 7: 100 episode average score averaged over 5 runs for SARSA(λ) with $\lambda = 0.9$ on Cart Pole

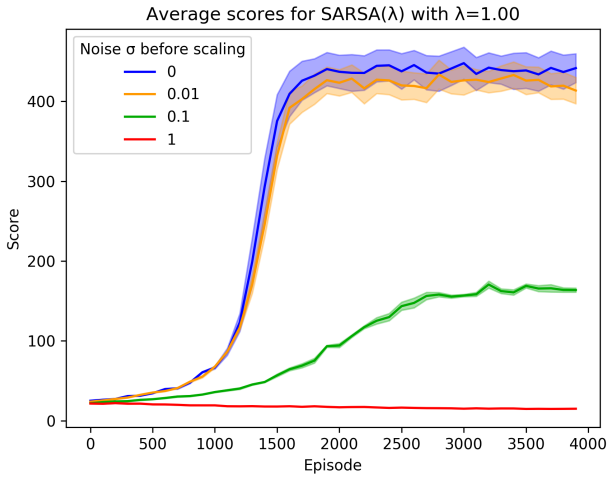

Figure 8: 100 episode average score averaged over 5 runs for SARSA(λ) with $\lambda = 1$ on Cart Pole

In Figures 6, 7 and 8 one can observe that higher values of λ do not have this problem. All 3 configuration reach > 400 scores reliably, although there are differences in stability between the different λ values. $\lambda = 0.75$ reaches the a score very close to the maximum score with very little variance, higher λ values appear to have a higher variance.

The results also show the impact of noise on the performance of SARSA(λ) . $\sigma = 0$ and $\sigma = 0.01$ often perform quite similar, although $\sigma = 0.01$ does seem to impair the performance slightly on average. $\sigma = 0.1$ massively impairs the agent but still allows it to balance the pole for roughly 150 timesteps. Note that this is a significant improvement over the random moves it does at the start of training, when ϵ is still close to 1. Choosing $\sigma = 1$ proves to be to much for the agent, as even a random policy outperforms the policy learned by the agent at this noise level.

3.1.2 Deep SARSA(λ)

The results for Deep SARSA(λ) are listed and discussed in this section.

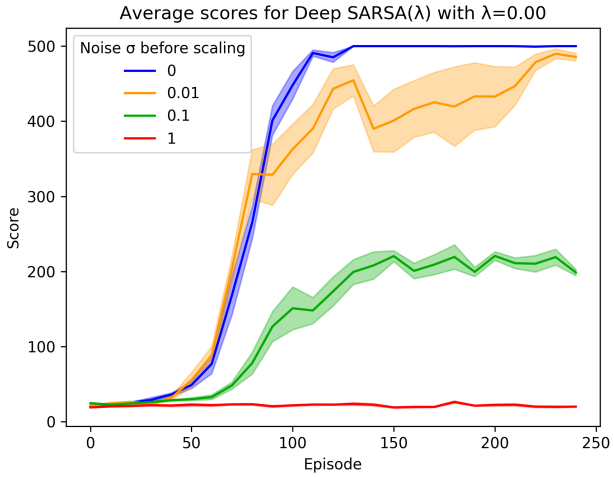

Figure 9: 10 episode average score averaged over 5 runs for Deep SARSA(λ) with $\lambda = 0$ on Cart Pole

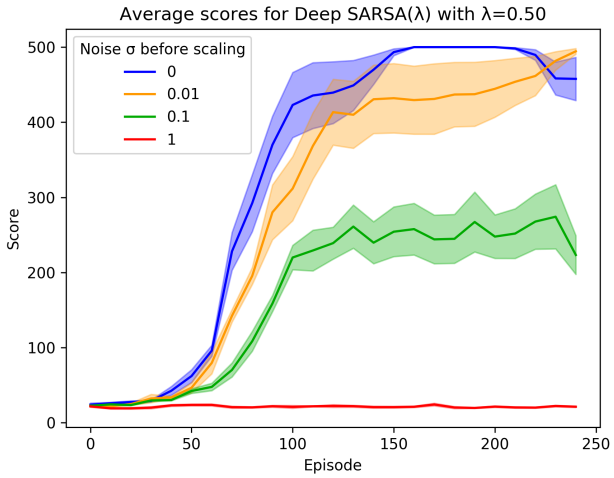

Figure 10: 10 episode average score averaged over 5 runs for Deep SARSA(λ) with $\lambda = 0.50$ on Cart Pole

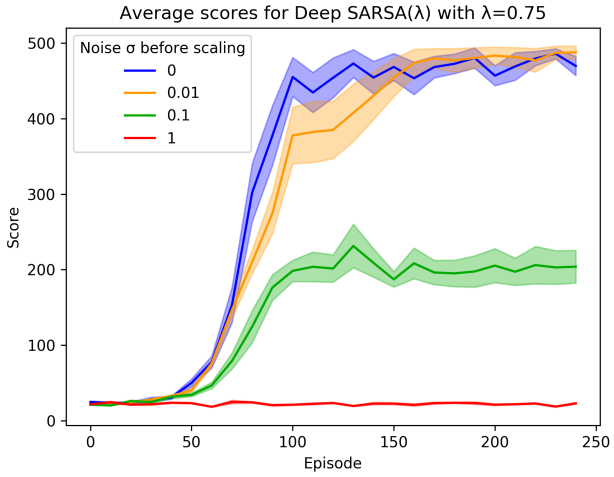

Figure 11: 10 episode average score averaged over 5 runs for Deep SARSA(λ) with $\lambda = 0.75$ on Cart Pole

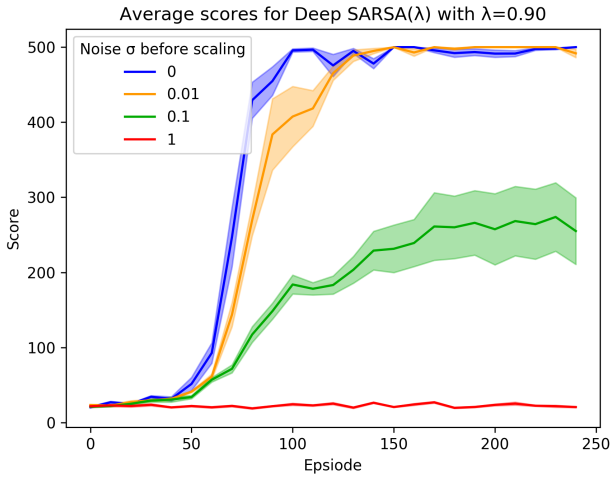

Figure 12: 10 episode average score averaged over 25 runs for Deep SARSA(λ) with $\lambda = 0.90$ on Cart Pole

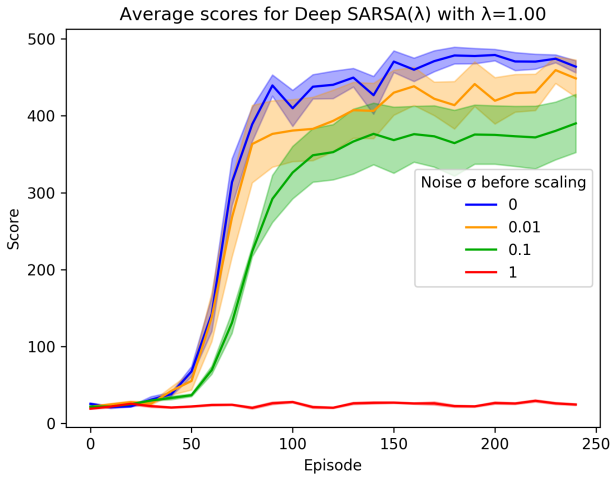

Figure 13: 10 episode average score averaged over 25 runs for Deep SARSA(λ) with $\lambda = 1.00$ on Cart Pole

In Figures 9, 10, 11 12, and 13 the plots for all λ values are listed. From these plots it becomes apparent that the neural network based method in combination with replay memory makes the system way more data efficient. SARSA(λ) required a setup with a more than 10 times slower epsilon decay to reach the same performance and therefore took more than 10 times as long to reach peak

performance. Interestingly enough, there does not seem to be a large difference between the different λ values in terms of performance. The only configuration that behaves differently is $\lambda = 1$, which is Monte Carlo. MC seems to deal with noise much better, but also performs relatively worse without noise compared to other values of λ. Overall, Deep SARSA(λ) appears to be slightly more resilient to noise than SARSA(λ) . This is mostly visible in the $\sigma = 0.1$ runs, where the average score appears higher or similar to SARSA(λ) on the same λ value. This effect is however not very large, apart for $\lambda = 1$. Note that due to the continuous state Deep SARSA(λ) is also able to learn at low λ values, something SARSA(λ) could not do due to discretization.

It should be noted here that there was an error in the implementation that mostly affected $\lambda = 1$. This implementation difference was fixed too late to rerun all experiments, so only $\lambda = 1$ and $\lambda = 0.9$ were redone. The new experiment for $\lambda = 0.9$ did not show any significant difference with the results listed here, leading us to the conclusion that the implementation difference did not meaningfully affect any λ value other than $\lambda = 1$ in this experiment. This is most likely due to the length of the trajectories being long enough to decay λ^k for these lambda values to zero before the end of the trajectory is reached.

3.2 Convolutional vs Deep Networks

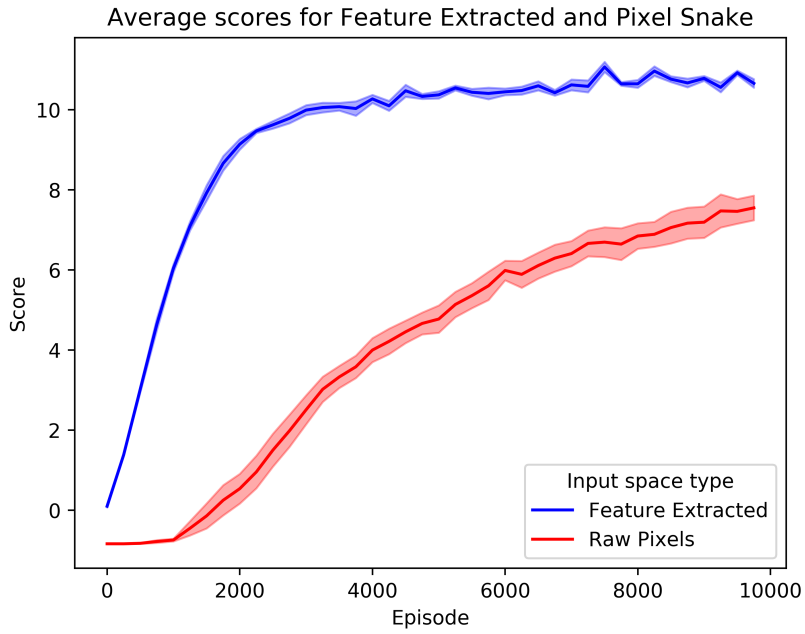

Figure 14: Snake with a deep network (feature extracted) vs Snake with a deep convolutional network (raw pixels). The lighter area depicts the 50%CI of the mean across the 5 runs.

The results of the comparison between deep and convolutional networks can be seen in figure 14. The darker lines represent the score of the results over 250 episodes and 5 runs whilst the lighter-shaded region represents the 50% confidence interval of the average between the 5 runs.

Beginning with the deep network operating on the feature extracted variant of snake's state, the score clearly approaches an asymptotic maximal performance. This asymptotic performance limit is likely a result of the combination of the network reaching its optimal performance and the ϵ reaching its minimum value ($\epsilon = 0.5\%$). This performance limit is near a score of 11, which because by definition of the episode having ended, means that snake ate 12 pieces of food. As the snake has an initial length of 3, the finial average length of the snake was 15. This is fairly impressive considering the grid is only 8 by 8 blocks and the snake only has access to the distance to collision in each of the aforementioned directions. The snake's body was occupying 24% of the grid's pixels.

The convolutional network operating on the raw pixel data did not reach convergence within the number of episodes tested. This is a result of the rate

at which the agent was improving being much lower than that of the feature-extracted deep variant. This is not a surprising result as the state-space of the raw pixels is significantly larger. This is further corroborated by the higher variance seen in the means of the raw-pixel variant as the curse of dimensionality means that the number of combinations of the state-space variables is drastically larger. It is desirable to train the raw-pixel variant for a larger number of episodes to determine whether increased expressivity of the state-space allows the network to train to a better result, but time did not allow for this.

It is the belief of the authors that the decay of the ϵ parameter is too large in the instance of the convolutional network. Due to the low variance and close-to-asymptotic performance of the feature extracted score, we believe the epsilon value to be very small by approximately episode 3000. It may, therefore, be the case that the convolutional network is being hindered by an excessively low epsilon value and thus not being sufficiently encouraged to explore the action space in as the number of episodes grows. This is particularly pertinent to the convolutional network as it will not have encountered exceptionally long snakes by the time ϵ is small and thus not learn well how to handle longer snakes.

Thus the hypothesis is not disproven, yet not proven either as the convolutional variant has yet to have a sufficient opportunity to possibly train to its asymptotic performance limit.

3.3 Value based vs Actor Critic

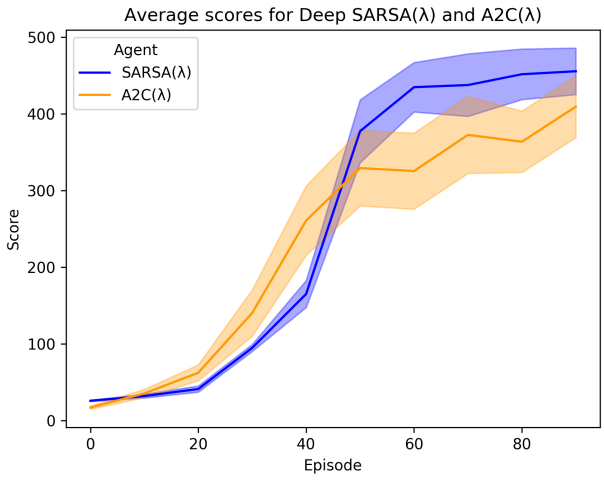

Figure 15: 10 episode average score averaged over 5 runs for Deep SARSA(λ) versus A2C on Cart Pole

In Figure 15 one can see a comparison between A2C and Deep SARSA(λ) . The results raise more questions than answers. First off, it seems like both

algorithms did not fully converge after 100 episodes, so A2C might actually converge to peak performance. Secondly, 2 runs for A2C and one run for Deep SARSA(λ) reached a score of 500, but then dropped down to a constant score of 10. Based off of these problems, we ran the tests a second time. This time longer and with 25 runs per algorithm to avoid the aforementioned issues. These results can be found in Figure 16.

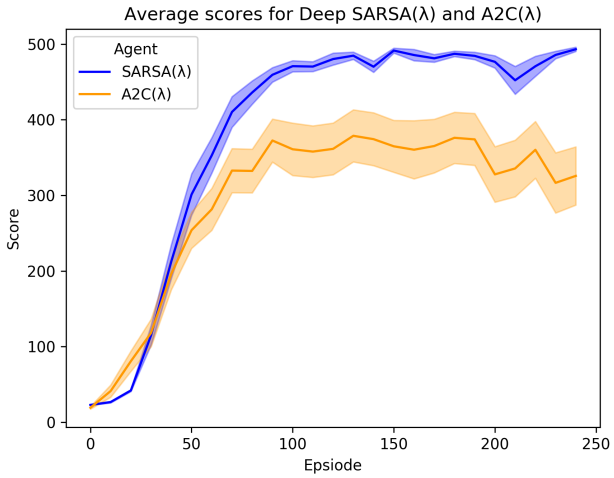

Figure 16: 10 episode average score averaged over 25 runs for Deep SARSA(λ) versus A2C on Cart Pole

These results show that the first image was not wrong or a statistical outlier. The A2C runs where highly variant, with some runs converging to the max score quickly and staying there, some runs converging very slow and still growing towards 500 and some runs crashing into the minimum score without recovering. These results do not show whether one method is definitely better than the other, but they do show that getting A2C in a stable configuration is harder than using only a value function approximator.

3.4 SAC-X

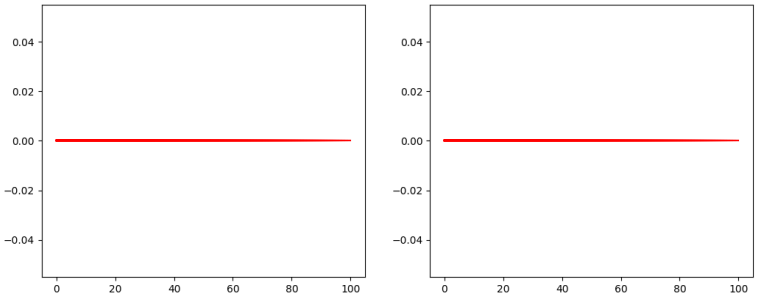

Figure 17: SAC-Q on Mountaincar, score/step (left) and average score on 10 last episodes (right) for the `MAIN` task.

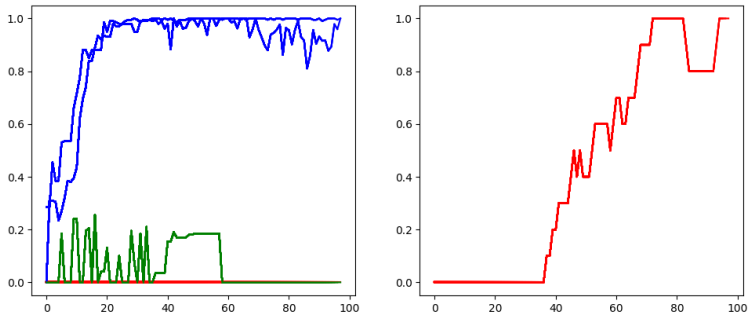

Figure 18: SAC-Q on Mountaincar, score/step (left) for each of the four tasks and average score on 10 last episodes (right) for the `MAIN` task. Red indicates `MAIN`, the blue scores/step are `GO_LEFT` and `GO_RIGHT` and green is the `GO_FAST` task

In Figures 17 and 18 results for applying SAC-Q on the Mountaincar environment can be found. Note that the plots on the left use scores from the previous episode if the task has not been scheduled this episode. From this one can see that without any auxiliary tasks the agent fails to learn the main task. When the agent does have access to 3 auxiliary tasks, it manages to perform the main task well after 100 episodes. One thing that should be noted is that although the agent manages to perform the main task well without having a good learned policy for this task. The high score is entirely gathered by using the `GO_LEFT`

and GO_RIGHT tasks. This is indicated by the two flat lines for MAIN and GO_FAST, which means that either these tasks have not been scheduled or the agent got 0 score per step on that task.

4 Conclusions

Draw more formal conclusions and add Conv and maybe some other conclusions. Also don't use itemize. Anyways, here's a summary of what we found:

- The field of Reinforcement Learning is immense and one cannot cover everything in 10 weeks.
- Neural Network based function approximation can provide an edge in both data efficiency (episodes) and performance in the environment, even in simple environments like Cart Pole. It does lose in wall clock time.
- While having a learned policy allows learning in environments with continuous action spaces, it is less stable compared to using ϵ-greedy in a discrete setting.

include in results

5 Discussion

- Q(s,a) at time 0 or time 500, for the same state, could be respectively 500 or 1. Which one is right?
- The tail of the weight distribution determined by λ is handled wrongly. In our implementation all "previous" weights are normalized.

5.1 Expected rewards in Capped Environments

When investigation the poor performance of Deep SARSA(λ) on the Cart Pole environment for $\lambda = 1$, we noticed an interesting problem for the algorithm. In environments like Cart Pole, where the episode length is capped and every timestep yields 1 reward, it can be hard to determine the value of $Q(s, a)$ in certain situations. When the agent performs better and better on Cart Pole, it will reach scores of up to 500. The cart and pole remain almost constantly in the same state, in the middle and without much velocity. This stable state in the middle can have an expected reward of 500 at the start of an episode, but also an expected reward of 1 at the end. The state isn't visibly different since the agent has no notion of time. So the correct estimate of $Q(s, a)$ can be anywhere between 1 and 500, which is not useful. Simple toy problems did show that this should converge to $Q(s, a) \approx 250$, although an inspection of the actual behaviour of our agents seemed to indicate that the Q-values were increasing continuously. Apparently this problem did not affect performance by much, as most agents managed to learn the Cart Pole environment, but it's still

an interesting problem to reason about. We did not have time to look further into this and thus cannot determine whether this causes the poor performance mentioned before.

5.2 Episodes vs Steps

When running the experiments, an interesting problem with the evaluation method was discovered. Due to episodes having a massively varying number of learning steps (10 - 500), episodes are not a good measure of time. 50 episodes on 10 steps are equal to 1 episode of 500 steps. Therefore an unlucky agent that needs 500 more steps to learn than another agent might just take 50 episodes more. Therefore comparing episodes till "solved" is a rather unfair method of evaluation in this case. It would have been more fair to cap the number of steps the agent could perform on the environment instead of the number of episodes. This issue also caused massive differences in runtime, where runs with bad configurations would end in minutes while the successful configurations went on for hours.

Acknowledgements

We would like to thank Dr. Christoph Brune for mentoring, supervising, and evaluating us on this work.

References

[1] D. Silver, "University college london course on reinforcement learning 2015." [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[3] M. A. Riedmiller, R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. V. de Wiele, V. Mnih, N. Heess, and J. T. Springenberg, "Learning by playing - solving sparse reward tasks from scratch," *CoRR*, vol. abs/1802.10567, 2018. [Online]. Available: http://arxiv.org/abs/1802.10567

[4] R. Munos, T. Stepleton, A. Harutyunyan, and M. G. Bellemare, "Safe and efficient off-policy reinforcement learning," *CoRR*, vol. abs/1606.02647, 2016. [Online]. Available: http://arxiv.org/abs/1606.02647

[5] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine Learning Proceedings 1990*. Elsevier, 1990, pp. 216–224.

[6] OpenAI, "A toolkit for developing and comparing reinforcement learning algorithms." [Online]. Available: https://gym.openai.com/envs/CartPole-v1/

[7] C. Beekhuizen, "Gym-like snake environment." [Online]. Available: https://github.com/av80r/Gym-Snake

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[9] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783