university of groningen / faculty of science and engineering / bernoulli institute

Backup policies for safe reinforcement learning in Mario

Bachelor's Project Thesis

Naut Folkers, s4377354, n.o.folkers@student.rug.nl
Supervisors: J.D. Cardenas Cartagena

Abstract: Is it possible to make a reinforcement learning agent that does not make dangerous mistakes when learning? The safe reinforcement learning field aims to enhance safety during the learning phase of RL algorithms. Experiments are performed on the influence of a backup policy using a one-step actor-critic algorithm to train on Super Mario Bros (NES). The goal is to return to a safe state when a critical state is encountered. A backup policy takes over when a critical state is encountered. This policy has an alternate reward function that prioritizes safety over level progression. Challenges emerge due to the misfit of the one-step actor-critic and the Super Mario Bros environment, such as policy collapse and incapacity to learn. Due to the lack of progression in the learning phase, it is impossible to enclose the effect of backup policies on the learning phase. However, a distinction can be drawn between safety reward functions in the backup policy. Rewards based on penalizing proximity to threats show more potential for threat avoidance when compared to rewards empowering maximum distance to threats.

1 Introduction

Reinforcement learning (RL) is a branch of machine learning (ML) algorithms that focuses on solving problems that require decision-making in dynamic environments. RL works by enforcing desired behaviors through rewards. These rewards are a consequence of the actions taken by an agent. The agent's goal is to maximize the reward it attains. To progress towards this goal, it learns to optimize the actions taken in given situations guided by these rewards. Multiple RL algorithms exist that implement this concept using different strategies, each with respective properties accommodated to the environment of deployment as described by Sutton and Barto (2020).

RL researchers have chosen video games as the test bed for their algorithms. Video games contain favorable conditions, such as episodic environments. Progressing through games presents distinct situations, such as checkpoints or in-game points, which are suitable as rewards for an agent. Additionally, games are repetitive and are optimal for gaining a large number of samples, which is valuable for model training.

Super Mario Bros is a 2D platform game for the NES (a game console created by Nintendo). The goal is to pass the levels by running towards the flag. Once the flag is reached, progress continues at other levels. In the final stage, Mario battles Bowser, his nemesis. The game has been so popular that new versions of Mario are still released.

Artificial intelligence (AI) advancements in the research sector have depended on available hardware's computational power. The shift to GPU computing made it viable to utilize deep neural networks (DNNs) as used in the state-of-the-art Artificial Intelligence algorithms today (Thompson, Greenewald, Lee, and Manso, 2022). DNNs are used in RL to learn more complex tasks that require more complexity to make decisions. Deep Q Learning (DQN) is a pioneer in deep reinforcement learning as it was one of the first algorithms to reach human-like performance in Atari games (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, et al., 2015).

RL can be tested in various games, such as 2D single-agent Atari games. Those games are a category for which Shao, Tang, Zhu, Li, and Zhao (2019) compared DRL algorithms against a human baseline. DQN can perform on a human level for 49 tested games in the ALE environment of Machado, Bellemare, Talvitie, Veness, Hausknecht, and Bowling (2017). Alterations have been made to DQN with the goal to enhance performance or to enhance sample efficiency. The PAAC algorithm performs sufficiently in games after a few hours of training. Results show that Ape-x DQfD bench-marked the best across 42 games with a mean of 2346% compared to human performance and a median of 702%. One of the strong points of the survey is the comparison across multiple games, this gives a

comprehensive overview of the algorithm's ability to perform in different environments. Compared to NES games such as Super Mario Bros, the Atari has limited graphics, meaning smaller DNN inputs.

1.1 Safe Reinforcement Learning

This project revolves around safe RL, which is a subset of RL. The differentiating factor is that models are not evaluated on their ability to solve a task but instead on how safely they can learn it. Safety is defined in this context as the ability to not end up in a fatal state. A fatal state is relative to the task and environment. For Mario, a fatal state would be to die because of an enemy. For an autonomous helicopter pilot, a fatal state would be crashing and destroying the helicopter, and an additional effect would be the discontinuation of the training. The difference between the two examples is that Mario can respawn, so the consequences are minimal. Contradictory, the helicopter crash is dangerous for the surrounding environment and would have significant cost consequences. Therefore, the philosophy behind safe reinforcement learning is to design safer algorithms using environments without real-world consequences to minimize the risk of consequences when employed in real-world environments.

Traditional RL algorithms use trial and error to learn the designated task. But trial and error comes with a significant risk of ending up in a fatal state. Minimizing this risk gives hope for RL algorithms to become a safe option for tackling real-world problems. An ideal safe RL agent will explore the possibilities for completing a task without the occurrence of a fatal state during training. To accomplish safety we can differentiate between 3 types of states: safe states, critical states and fatal states. A safe state is a state that poses no threat to the safety of the agent. A critical state is a state that can transition into a fatal state but it is still possible to transition back to a safe state, in short, the agent is in danger but can avoid elimination. A fatal state is a state in which there is no transition possible back to a safe state, and failing due to the opposing danger is imminent. To guarantee the agent's safety, it is necessary to identify when an agent is in a critical state to avoid the fatal state and transition back to a safe state (Hans, Schneegaß, Schäfer, and Udluft, 2008).

One of the challenges of RL is the balance between learning (exploring) and using what is learned (exploiting). Exploring consists of taking actions on a random basis to find which actions yield the maximum rewards for a given situation. After the optimal action is found, the agent should exploit this knowledge to gain maximum rewards for this situation. When an agent mainly explores, it results in a lack of progress. On the other hand, doing too much exploiting will restrain the agent from performing optimally. This balance becomes even more critical for safe RL because exploration can result in unforeseen fatal states. However, being too cautious of unforeseen fatal states will stale the agent's progress. A conservative agent is better than a curious agent because guaranteeing training continuation is better than exploring too bold and resulting in fatal states.

The challenge we set for this project is about finding a way to improve RL safety during training through the Super Mario Bros Gym environment (Kauten, 2018). A safety factor will be induced into the training of our agent. The experiment will track the death count of our agent to measure whether the safety factor has the intended effect. The experiment draws inspiration from the safe RL strategies presented by Garcıa and Fernández (2015). Two categories are presented for improving safety: alternation of the optimization criterion and alteration of the exploration process. A regular RL agent's optimization criterion would be maximizing the expected reward. However, in this category, alterations are made to this criterion to incorporate risk, uncertainty or constraints to the policy to influence the learning toward a safer agent. Alternatively, the exploration process can also be altered by giving the agent prior knowledge. This prior knowledge is provided to help an agent avoid apparent mistakes. The agent is less likely to reach a fatal state when the exploration actively avoids dangerous parts of the state space. The methodology in section 3 will present our exact implementation of the experiments that include these strategies.

1.2 Related Work

In Liao, Yi, and Yang (2012) an agent is built using the Q-learning algorithm in a community adaptation of Super Mario Bros containing AI testing tools created by Adam Dingle (2012). An overlay grid is used to determine different zones of the screen. These different "windows" act as environment descriptions to convey to the agent whether enemies are present in the specified windows. A replay buffer is used to store states which helps the agent remember previous states. There is also an environment variable that tells the agent whether there is an obstacle in front of him. However, there is no direct safety directive for the project.

Guo, Yu, Lan, and Jin (2023) experimented with advantage actor-critic in Mario with the goal of making RL more interpretable. Research is done on the effect of a reasoner component that could classify the progress of Mario during an episode. A hamming window with the shift theory of 2D

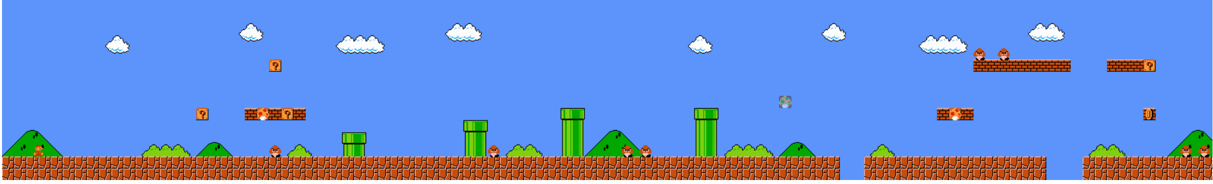

Figure 1.1: Mario world 1-1 (NesMaps.com)

DFT is used to find the difference between game frames with seemingly good results. It is concluded that a reasoner component added to the actor-critic improves visual interpretation and progress classification. It would be relevant to this project to see the reasoner's performance in the classification of critical state avoidance done by an agent.

2 Theoretical Framework

2.1 Reinforcement Learning

Reinforcement learning can be formalized in Markov Decision Processes (MDP) as described by Sutton and Barto (2020). It is a mathematical formulation for the agent-environment interaction that RL problems consist of. In finite MDP's there is a finite set of states $\mathcal{S}$ containing all possible states in the environment. A finite set of actions $\mathcal{A}$ contains all actions in the action space. A reward function $\mathcal{R}$ where: $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}' \rightarrow \mathbb{R}$. And the transition function $\mathcal{P}$ that represents the transition probability between states where: $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$. The MDP can then be described as a set $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. In addition to this mathematical formalization of the interaction, RL adds a goal for the agent: maximizing the cumulative reward it receives. Instead of maximizing immediate reward, which makes an agent shortsighted, the goal is to perform well overall, which is why future reward should also be considered. For this, the discount factor γ is used where $\gamma \rightarrow [0, 1]$. The discount factor is used to discount the expected future reward, this will decrease the value of a future reward when compared to the immediate reward. Using γ values closer to 1 will result in greater expected future reward values. Therefore, future rewards are considered more valuable in this situation compared to when γ values closer to 0 are used, which would decrease the expected future rewards to lower values. How an agent interacts with the environment is based on the policy. A policy is responsible for deciding the action based on the state: $\pi : \mathcal{S} \rightarrow \mathcal{A}$, where π is the policy of the agent.

2.2 Deep Reinforcement Learning

Conventional RL can be extended to use deep learning techniques to learn more complex tasks. Before deep learning techniques, RL had difficulty with large feature inputs such as visual inputs that consist of thousands of pixels. The ability to extract deeper abstractions from input data as is possible with DNN's improves the capabilities of RL. Because the agent in Mario receives frames as input, these techniques are required to make abstractions of the vastly dimensional state space of Super Mario Bros.

2.3 Actor-Critic

Actor-Critic is an algorithm that combines Value-based and Policy-based methods to get the best of both. The actor is based on a policy gradient algorithm that utilizes the parameters of a network to derive a policy directly. The critic utilizes V-values originating from value-based algorithms to produce a value estimation of the state (Francois-Lavet, Henderson, Islam, Bellemare, and Pineau, 2018). As stated by Konda and Tsitsiklis (1999), policy-based methods are considered actor-only algorithms and value-based methods are considered critic-only algorithms. Policy gradient methods suffer from high variance because they lack memory replay and use a probability distribution for action selection which adds stochasticity. Critic algorithms may succeed in learning good approximation functions but these are described as "indirect" because an optimization problem needs to be solved to derive a policy. These value-based methods are required to maximize Q-values and V-values, when an environment contains a large state space, this optimization problem is difficult to solve (Sutton and Barto, 2020). An actor-critic algorithm aims to use the strong points of both of the algorithms to perform better overall. It uses a critic to learn an approximation of a value function, which gives an informed estimation based on previously encountered states. This value function is then used to update the actor parameters. The actor implements the policy directly in its parameters to eliminate the indirectness of only using function approximation.

The actor-critic algorithm used for this project is the one-step actor-critic algorithm from Sutton and Barto (2020) as seen in Algorithm 2.1. It is an online version of actor-critic, so the parameters are updated for every step. It uses Temporal Difference

Algorithm 2.1 One-step Actor-Critic (episodic), for estimating $\pi_\theta \approx \pi_*$, (Sutton and Barto, 2020)

Require: a differentiable policy parameterization $\pi(a|s,\theta)$
a differentiable state-value function parameterization $\hat{v}(s,w)$
step sizes $\alpha^\theta > 0, \alpha^w > 0$
Initialized policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 0)
for each episode **do**
Initialize S (first state of the episode)
while S is not terminal (for each time step) **do**
$A \sim \pi(\cdot|S,\theta)$
Take action A, observe S', R
$\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$ (if S' is terminal, then $\hat{v}(S',w) = 0$)
$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S,w)$
$\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(A|S,\theta)$
$S \leftarrow S'$
end while
end for

(TD) to calculate δ, which is used to calculate the loss of the critic and the actor. This means that the discounted expected reward is compared to the actual reward gained to learn the estimation error from the value function. Different loss calculations were used for the actor and the critic. The critic loss was calculated using the mean squared error of the temporal difference as shown in (2.1).

$$Loss_{critic} = (\hat{v}_w(s) - (r + \gamma\hat{v}_w(s')))^2 \quad (2.1)$$

Where $\hat{v}$ is the value function, r is the reward gained in the last step, s is the current state and s' is the next state, w is the current parameterization of the critic and γ is the discount factor. The actor loss is calculated using the TD, a decay factor, and the negative logarithmic probability of the action taken under the current policy as shown in (2.2).

$$loss_{actor} = -\ln\pi_\theta(a|s)(\hat{v}_w(s) - (r + \gamma\hat{v}_w(s')) \quad (2.2)$$

Where r is the reward gained in the last step, θ is the parameterized policy of the actor, and π is the current policy. Additionally, there is the shared loss, which is the sum of the actor loss and the critic loss.

3 Methodology

3.1 Mario Environment

To run experiments, the Gym Super Mario Bros environment from Kauten (2018) is used, which is based on the RL environment library called Gym by Brockman, Cheung, Pettersson, Schneider, Schulman, Tang, and Zaremba (2016). This environment allows us to run Mario in a controlled step-wise manner. It implements the Mario game and provides a way to communicate with the environment. Several down-sampled versions of the game can be chosen, "V2" was used for the experiments because it allows for enemy detection and is easier to interpret for a convolutional neural network, as displayed in Figure 3.1. The experiments were performed in World 1-1 as this was enough of a challenge, and switching between worlds or levels during training meant that the colors in the frames changed, which could make it harder for the agent to recognize its environment. Tools in the environment equip the agent with step-wise action selection. It is equipped with a reward function. Rewards consisted of:

- Positional reward: the difference in Mario's position between steps in pixels to the right is given.
- Clock reward: the negative difference in clock reading between steps is given.
- Death penalty: for dying the agent receives a death penalty of -15.

To clarify, the agent receives a negative reward for moving to the left, 0 for not moving and a positive reward for moving to the right. For every tick on the clock that goes down in a step, the agent receives a negative reward, according to the environment documentation this is to prevent standing still. The maximal and minimal reward an agent can receive from the environment is clipped between (-15, 15). Finally, the relevant status updates received from the environment after taking a step include: whether Mario finished the level, the number of lives left, the x-position in the level, the y-position in the level, the x-position from the left side of the screen in pixels. The last x-position is not available in all environment versions but is easily added by a slight modification in the library.

Mario's action space is reduced to two inputs. The set of actions: $\mathcal{A} = \langle\langle Right\ \rangle, \langle Right + Jump\ \rangle\rangle$. These actions are the two only options required by the agent to finish the level.

The state space $\mathcal{S}$ is defined such that $s \in \mathcal{S}$ is a 3-dimensional tensor. This tensor has dimensions $4 \times 84 \times 84$ with a normalized pixel value between 0 and 1. Each state $s \in [0,1]^{4\times84\times84}$, where 4 represents the 4 frames stacked together and 84×84 are the height times width of each frame.

3.2 Preprocessing

The input the agent receives from the environment is preprocessed frames from the game. Firstly, the

Figure 3.1: A frame displaying the starting position of Mario in the "V2" simplified version of the environment

environment is wrapped to skip frames. It is implemented so that four frames are managed during a step. The rewards gained during these skipped and observed frames are summed into one reward that is returned after each step. The second step of preprocessing is a grayscale wrapper. This applies to observations, its colors are mapped to grayscale. This reduces the color dimensions of the observation from 3 dimensions to 1 dimension. After this, the observations are resized to reduce input dimensions. The frame's dimensions are downscaled from 256×240 to 84×84. During this step, the pixel values are normalized from a 0-255 range into a 0-1 value range. Finally, the four frames that occurred during the step are combined into one observation, from this observation a sense of direction can be observed by the agent. To illustrate, when the agent jumps, the first frame in the observation would still display Mario on the ground, while the last frame would show Mario in the air. An example of a processed observation from the environment can be seen in Figure 3.2.

3.3 Model Architecture

The implementation of the network was done in `PyTorch` the library from Paszke, Gross, Massa, Lerer, Bradbury, Chanan, Killeen, Lin, Gimelshein, Antiga, Desmaison, Kopf, Yang, DeVito, Raison, Tejani, Chilamkurthy, Steiner, Fang, Bai, and Chintala (2019) using `Python 3.10.4`. The GPU-accelerated version of `PyTorch` was used to run the network from a GPU to speed up the training. The model used is a combination of a shared layer part consisting of convolutional layers and linear layers. On top of that, there are two separate heads, one for the critic and one for the actor. Outputs from the shared layers are forwarded through both of the heads but the output of both heads does not influence the other head. Table 3.1 gives a technical summary of the architecture. A graphical construction of the network to illustrate the flow from the shared layers into the individual heads of the actor and the critic can be seen in Figure 3.3.

Under parts of the network, three parts are considered: the shared layers, the actor head and the critic's head. all parameters of these parts are updated at once using the shared loss. The shared loss is the sum of the actor's and critic's losses. These losses are optimized by back-propagating through both heads and the shared layers at the same time.

3.4 Experimental setup

Safety in RL is the focus of this project, therefore an experiment was done to see if the learning phase could be made safer in Mario. There are three ways the agent (Mario) could die during the level:

- Walking into a Goomba
- Walking into a Turtle
- Falling into a pit

Prior knowledge is induced into the agent by implementing a detector for the Goomba and the pits. Turtles were not considered for this experiment. Critical states were marked when the agent was within a specific ϵ-distance range of an enemy. The experiment is called backup policies because the agent would have a policy that was used only when in a critical state, so near a goomba or a pit. The idea is to have a normal policy that tries to finish the level and a policy that tries to keep the agent away from fatal states. In this distance, the reward is modified to become a safety reward. Experiments were done with two modified safety rewards, a positive version and a negative version. The positive version gives more reward when the distance is greater within the critical range of ϵ, with the idea of promoting distance (3.1). The negative version gives a negative reward proportional to the inverse distance within the critical range of ϵ, with the idea of penalizing proximity to threats (3.2).

$$SRpos = (1 - \lambda)R_{normal} + \lambda * Distance \quad (3.1)$$

$$SRneg = -\epsilon + (1 - \lambda)R_{normal} + \lambda * Distance \quad (3.2)$$

SR is the safety reward denoted by *pos* for the positive version and *neg* for the negative version. λ is a balance factor to balance the ratio between normal reward and distance as part of the safety reward. R_{normal} is the reward that is gained from the environment. *Distance* is the pixel distance

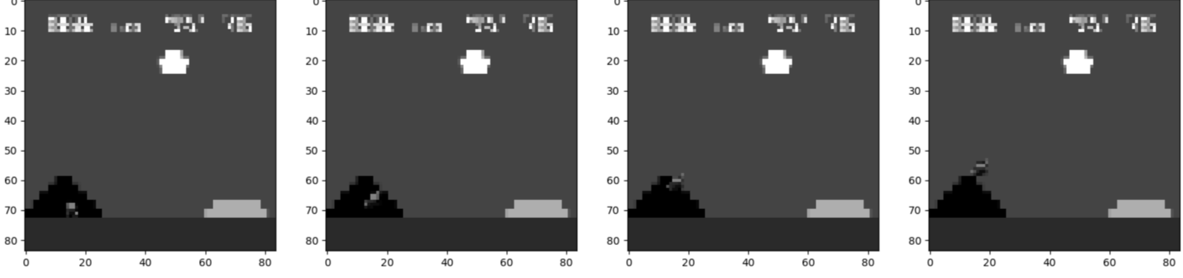

Figure 3.2: Example of a preprocessed observation.

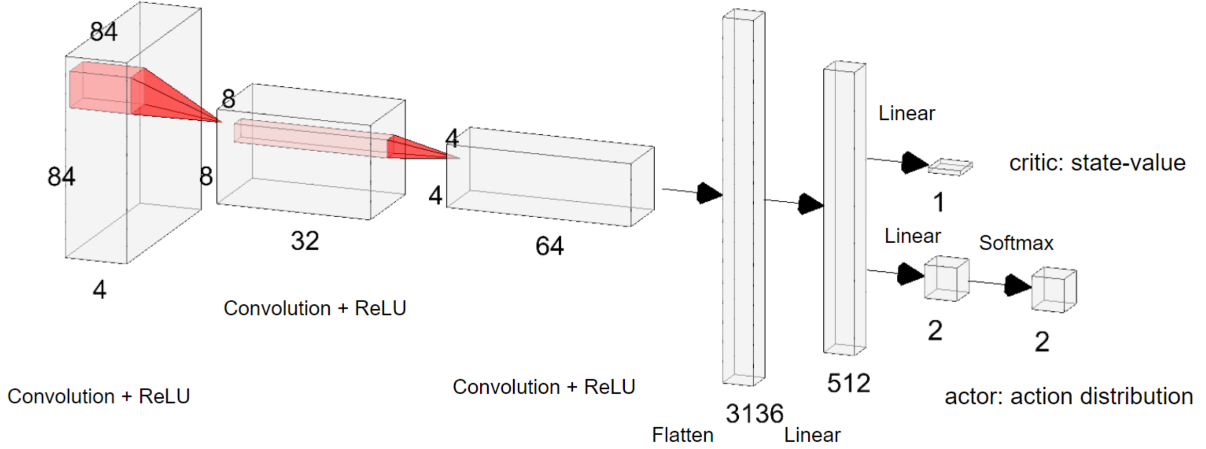

Figure 3.3: Model architecture for the Actor-Critic agent

measured by the detectors between the agent and the closest threat in Euclidean distance:

$$Distance = \sqrt{(x_{mario} - x_{threat})^2 + (y_{mario} - y_{threat})^2} \tag{3.3}$$

3.5 Enemy & Distance Detection

To identify critical states and initiate a policy switch, detectors must be used to find and extract a distance from the environment. Part of the information is accessible through the Mario environment. At every step, info is returned where the `y_pos` and `x_left` keys are used to extract Mario's position in the frame. Because the size of the frames was reduced these values have to be scaled accordingly, equations (3.4) and (3.5) are used for this.

$$ypos_{new} = \frac{ypos_{old} - 32}{3} \tag{3.4}$$

$$xpos_{new} = 2 + xpos_{old} * 0.35 \tag{3.5}$$

In (3.4), the number 32 is subtracted, this value was found to be added to the vertical position of Mario as registered in the environment. This was not the case visually, therefore it had to be accounted for before scaling the position. For (3.5), the scaling factor to go from 240 pixels to 84 pixels is 0.35.

When `x_left` value is disabled, it must be activated manually so that it returns this private member through the info function of the environment. After acquiring Mario's horizontal and vertical positions, they are used as Mario's coordinates for distance measurement.

3.5.1 Goomba's

Goomba's are the first threat that Mario encounters in the level. To detect them, the frame of the current state is cut so that only the relevant parts remain. Therefore, the ground (11 pixels high) and a part of the sky (24 pixels high) are cut off, this way the search area is reduced. Analysis of the frames showed that the feet of the goomba contain a unique color. The cut frame was masked so only pixels between the normalized color range $[0.76, 0.77]$ remained, and all other values were set to 0. The remaining color is the exact unique color found on the goomba. To extract coordinates from the frame the `nonzero` function from `PyTorch` was used. These coordinates are checked to see whether they have a horizontal neighbor. In case there are horizontally neighboring coordinates, a goomba is detected. If there are no neighbors or vertical neighbors, it is a false detection of an anti-aliased

Table 3.1: **Pytorch Architecture of the Neural Network**

Layer	Type	Input	Output	Kernel Size	Stride	Part
1	Conv2d + ReLU	4×84 × 84	32×20 × 20	8×8	4	Shared
2	Conv2d + ReLU	32×20 × 20	64×9 × 9	4×4	2	Shared
3	Conv2d + ReLU	64×9 × 9	64×7 × 7	3×3	1	Shared
4	Flatten	64×7 × 7	3136			Shared
5	Linear + ReLU	3136	512			Shared
6 Actor	Linear + Softmax	512	2			Actor
6 Critic	Linear	512	1			Critic

Figure 3.4: Processing pipeline for goomba detection. In short, first cut the frame, then mask for certain pixel values, and lastly, extract coordinates.

piece of the bush from the background. Finally, all detected goomba's Euclidean distances to Mario are compared. The smallest distance is compared to the ϵ-distance for critical state evaluation. A visual representation of the process is displayed in Figure 3.4.

3.5.2 Pits

Pits are the second obstacle found in the game, and it is difficult for an agent to avoid falling to its death. To detect pits the frame is cut so that only the two top pixels of the ground remain (counted from the bottom the 10th & 11th pixel). Only the pixels containing higher x-coordinates than Mario are considered. Mario can only move right so considering passed pits is unnecessary, so the left side is omitted. The ground in the environment always has the same color value. Therefore, that normalized color is subtracted from the cut frame: $Frame - 0.4863$. If all pixels are now 0, which is black, then it means there is only ground in this part of the frame, so there is no pit. If there are nonzero pixels, there is a pit, and the coordinates can be extracted again using the `nonzero` function as used before for the goomba's. The closest part of the pit is extracted first, so no comparison has to be made between the pixels' distances to Mario. From this first coordinate, the Euclidean distance is compared to the ϵ-distance for critical state evaluation. Figure 3.5 displays a visual summary of this process.

Once either a pit or a goomba is within ϵ-distance of Mario, a flag is raised that switches to the safety policy. The agent does not receive the input as a feature but due to the policy switch, the rewards are guiding the agent to avoid threats instead of level completion.

3.6 Hyperparameters

It is essential to tune hyperparameters to maximize the results of an agent. A lot of tuning was done and the hyperparameter values used for our experiments are presented in Table 3.2. The learning rate was adjusted manually by checking the training data on test runs. Due to the computationally heavy aspect of this project, it was not manageable to run a grid search across hyperparameter space. The main criterion for tuning the learning rate was avoiding policy collapse. Using greater learning rates caused the policy to collapse more often and also earlier. Taking a smaller learning rate means slower learning times, and since this project suffers from a lack of learning ability, it was questionable to pick such a small learning rate. However, a greater learning rate (0.0001) with more episodes (40000) did also not show the ability to learn, therefore it was decided to minimize collapses. The discount factor (γ) was also hand-picked, different values

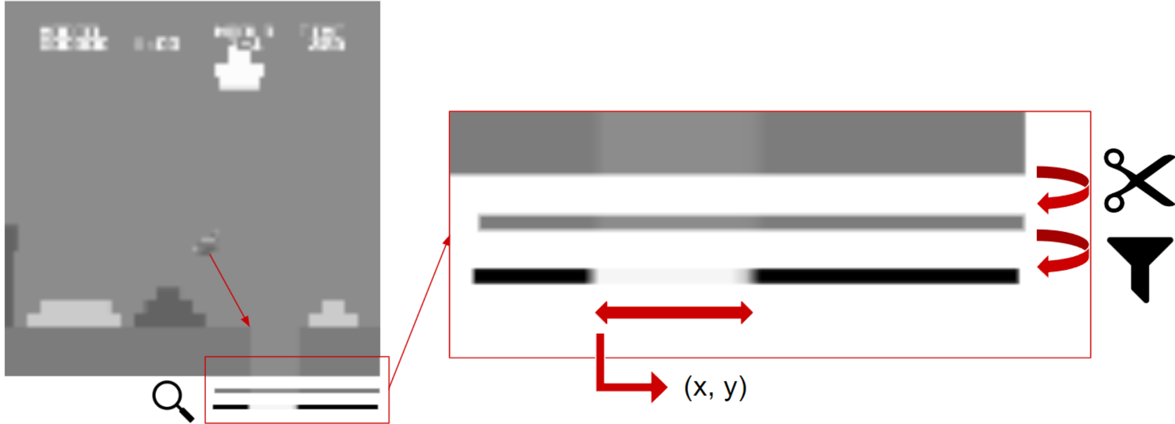

Figure 3.5: Processing pipeline for pit detection. In short, first cut the frame, then subtract ground pixel values and lastly, extract coordinates.

Table 3.2: Hyperparameters

Hyperparameter	Value
Learning rate	0.00000001
discount factor (γ)	0.99
ϵ-distance	20 pixels
safety reward balance factor (λ)	0.99
Environment	V2
Level	1-1
Action Space	[[`right + jump`], [`right`]]
Frame skips	4
Lifes	3 (standard)
episodes	15000

ranging between 0.85 and 1 were tested and it was found that the agent would get stuck earlier in the level for lower values. Therefore 0.99 was chosen, this value is used in the field often. A requirement for the ϵ-distance is that the safety agent always has enough time to avoid threats. To find this range a heuristic of prerecorded inputs was created that maneuvers around the threats from the moment it was activated. Then, the ϵ-distance was tuned upwards from a value of 7 pixels until the point where it would never fail to avoid the threat in time, this distance is 20 pixels in Euclidean distance. For the balancing factor of the safety reward (λ), it was found that an increased influence of the safety reward caused more change in behavior by the agent. Environment "V2" was picked because "V0" and "v1" were slower to run and "V3" was simplified to a point where no goomba detections were possible. Level 1-1 was used because goomba's were distinguishable and it is easier than other levels. 15000 episodes were used because this was around 24 hours of training time. The parameters required a lot of testing which is not manageable to do when an experiment takes four days to run, tests were done on longer experiments but the trends did not change much in the last 20000 episodes of a 40000 episode experiment, except for sudden collapses.

4 Results

The experiments were intended to test whether a switch in policies made the learning phase safer in terms of fatal states. However, the agent did not show an upward trend in reward for any of the situations tested. The configurations were as follows:

- An agent with only a normal policy.
- An agent with only a safety policy that took over from a random agent.
- An agent with both a safety and a normal policy that switches depending on the state status.

The active policy depends on the agent's distance from a threat. When Mario is within ϵ-distance of a threat then the safety policy is active. When Mario is outside ϵ-distance to a threat, the normal or random policy is active depending on which configuration is running.

$$\pi(a_t|s_t) = \begin{cases} \pi_{normal}(a_t|s_t) & distance > \epsilon \\ \pi_{safety}(a_t|s_t) & distance <= \epsilon \end{cases}$$

In this equation, π is the active policy. a is an action such that $a \in \mathcal{A}$, s is a state such that $s \in \mathcal{S}$. Lastly, t denoted the current time step. When the $safety$ policy is used together with a random agent, this random agent replaces the domain of the *normal* policy.

Figure 4.1 shows an overview of these different configurations. Both policies that use a backup policy were tested on a negative and a positive safety reward function. The only signs of improvement in policy were found in the normal + negative safety policy agent and the normal-only agent. However, these signs are observed as a trend in the logged

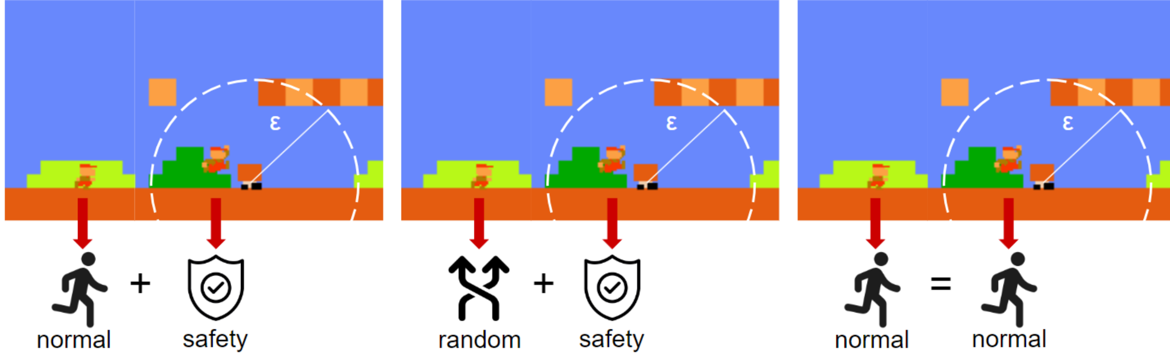

Figure 4.1: An overview of the three combinations of policies used in the experiments.

data but did not show an improvement in the ability to play the game. For the normal agent, it is observed that it starts with a base reward level. Later it unlearns to acquire this reward level, and then relearn its previous base level again. Therefore, it could be stated that there was a form of learning but not an improvement in ability. The reward is observed to decrease for the normal + negative safe agent. However, it is the only experiment run where there is an improvement in the survival rate of critical states. Adding to this observation, it must be stated that the level at which it learns to survive critical states is similar but higher than the base level of other agents. These other agents did not show to have learned this but might have had favorable initial runs in which they managed to survive by almost random action. An experiment was run on a completely random agent and it showed to be unexpectedly good. Because the action space only contains a form of running or running and jumping simultaneously, the random agent jumps about half the time, which is a good strategy for Mario. It can be speculated that this partially explains problems encountered during other runs of the experiment.

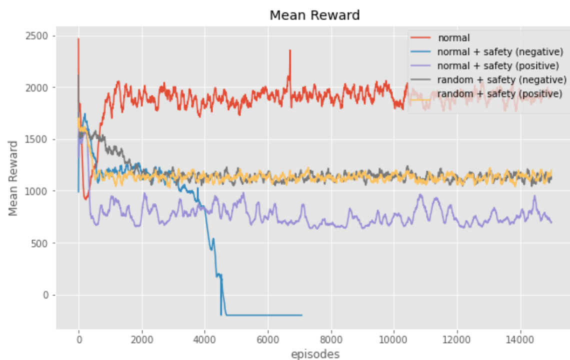

Figure 4.2: Average reward per episode compared by agent configuration.

The agent seemed to have trouble with improvement, but other problems were observed as well.

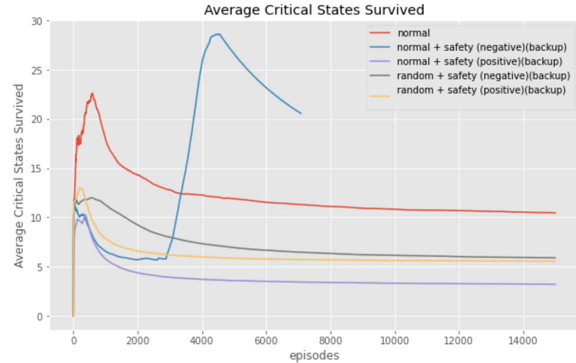

Figure 4.3: Average critical state survivals per episode compared by agent configuration.

Figure 4.4: Average Safety reward per episode compared by agent configuration.

When the experiment runs were extended with more episodes there were either two outcomes. Outcome one: the agent did not improve and did also not recede, progress in terms of reward was horizontal. Outcome two: the agent had horizontal progress, until the sudden collapse into a bad policy. In the second case, the evaluation of the model shows that the agent gets stuck at the first or second green pipe. The probability distribution has shifted towards one of the two inputs. The other action has a probability of around $\pi(a|s) = 1.212 \times 10^{-10}$ to be selected. In the case where `right + jump`

was the action chosen all the time, it would cause an inability to jump. This is because after landing a jump it is required to let go of the jump button first to be able to jump again. If no pause in jumping occurs, the effect of `right + jump` after the first jump is the same as `right`. In the other case of `right`, the agent is guaranteed to die or get stuck on the next obstacle. Efforts to prevent these outcomes include: tuning the learning rate up and down, tuning the discount factor up and down, switching between model optimizing methods, trying `SmoothL1Loss` for the critic, using `xavier_uniform` algorithms for the initialization of the network parameters, freezing shared layer parameters, using different action spaces, using a different down-sampled version of the environment, enabling and disabling the discount factor (I from Algorithm 2.1) and prolonged experiments. An interesting observation shown in Figure 4.6 is that the total death count between the agents was identical except for the normal + safety (negative) agent. This agent learned to get stuck on a pipe, it can be argued that this is the result of a negative reward when a threat is encountered further in the level. Its behavior is to jump over the first goomba and then run against a pipe until the timer runs out.

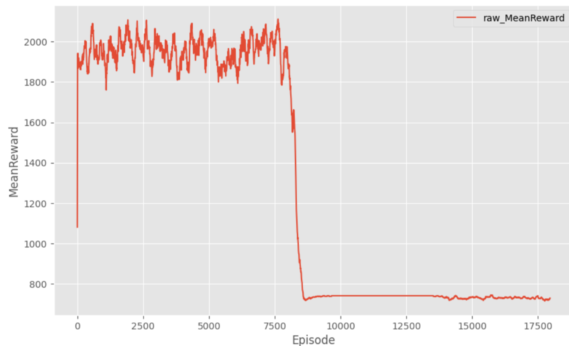

Figure 4.5: Collapsing policy problem.

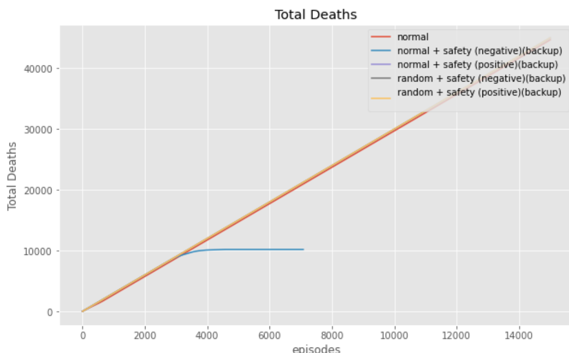

Figure 4.6: Total deaths compared to episodes. The normal + safety (negative) agent avoids death by getting stuck and running out of time.

Tuning the learning rate down delayed policy collapse or resulted in outcome one. This can be explained because it takes more wrong steps during back-propagation to reach another policy as opposed to a higher learning rate where a smaller amount of updates could result in a model with a collapsed policy. Freezing shared layer parameters had the same effect but reservoir computing (Lukoševičius and Jaeger, 2009) was not intended to be part of the project but rather an experiment on whether it affected learning. An explanation for this is that the shared layers are a large part of the architecture, if learning is disabled in these parameters the collapse would take longer to arise from only updating the parameters of the heads. The other efforts showed no notable effects, policy collapse occurred even earlier or outcome one or two was observed. Another potential explanation for the policy collapse is the balance between actor loss and critic loss. Konda and Tsitsiklis (1999) stated that the actor parameters might need to be updated on a slower time scale than the critic parameters in case problems arise. Two methods of model optimization were tested, one in separation and one over all the parameters at once. When the parameters are all updated at once, there is no control over the learning rate of the individual heads. Secondly, tuning the balance of their respective learning rates tends to be complicated. Analysis of the logged data from the experiments shows that the critic loss is way higher than the actor loss which means that most of the layers are only updated on the loss of the critic, the actor loss is too small to impact the total loss.

These results conclude that this implementation of actor-critic is not a good fit for the assigned task. Its instability during learning makes it difficult to get reliable results on the intended backup policy experiment. When an agent cannot learn the task, it has to be taken into account that this could also be the case for the safety task. Unfortunately, because of the inconsistent and unstable learning behavior that is observed, it is difficult to test the cause of the issue. In some cases, two experiments with the same agent configuration and hyperparameters differ a lot in the resulting policy. The only consistent result observed was that there was no experiment run where the agent made progress toward earning more rewards.

5 Discussion

This research should conclude whether a backup policy makes the training of a RL agent safer. To derive an accurate answer, it is necessary to compare the learning of an agent with a backup policy and the learning of an agent without a backup policy. This is not what the results show, the results

show that the agent is not capable of improving its policy under the conditions used during the experiments. Concluding the research question is not possible, but differences are observed during the experiments. Positive safety reward functions do not show differences when compared to non-backup-policy agents in the used setup. The agent with a negative safety reward function shows more potential for further research because a distinct pattern was observed from the training data. It showed an improvement in average critical state survivals. The learning instability made it hard to get consistent results for hyperparameter tuning. Because this environment was computationally heavy to run, performing an automatic grid search over hyperparameter space with the resources at hand was impossible. Issues with server availability have made it increasingly challenging to conduct experiments to improve learning ability. When an agent updated a policy such that it got stuck at someplace in the level, it increased episode times. These increased episode times resulted in the early termination of the experiment due to server limitations. This also explains the inconsistent episode count of the graphs 4.2, 4.3 and 4.4, shown in the results section.

Due to the unforeseen learning implications, it is now more important to analyze what caused the instability during learning. As mentioned earlier, the balance between actor and critic learning rates is a concern for the problem. Comparison between the actor loss and the critic loss shows that the actor loss is virtually not contributing to the total loss. Actor loss starts in a range from -4 to 8 and converges to a range between -1 and 1. Critic loss starts in the range from 0 to 200 and explodes to a range between 0 to 4000, depicted in Appendix A (Figures: A.1, A.2 and A.3). This tells us that the critic is contributing massively to the parameter updates of the model. This could further destabilize the learning when the gradient descent overshoots the desired direction. The learning rate used was small to mitigate the effects of this phenomenon. This resulted in delayed or complete avoidance of policy collapse. However, a small learning rate greatly affects the sample amount needed to learn and the signs of mitigated learning combined with a small learning rate are not ideal.

Using only an individual environment to learn from also makes learning unstable. This is why the A3C algorithm updates the policy based on the performance of multiple environments (Mnih, Badia, Mirza, Graves, Lillicrap, Harley, Silver, and Kavukcuoglu, 2016). In their paper, it is stated that online RL algorithms are unstable without utilizing batched inputs. To make the algorithm online without experience replays, a collection of asynchronous environments is used to learn from. The distribution of experience over multiple scenarios makes the weight updates more robust. Our problems would benefit from this idea because the policy collapses after a series of bad updates. These bad updates are based on the rewards obtained from only one environment. By basing the algorithm for Mario on a more stable algorithm it is easier to see the full effect of a backup policy when it is not hindered by the stochastic outcomes of only one environment, averaging multiple stochastic outcomes brings the policy to a better-suited update. Li, Bing, and Yang (2018) also states that the correlation of the sequential data is a cause of instability for on-policy actor-critic algorithms.

The shared layers in the network architecture have the purpose of learning patterns in the visual input data. However, the agent seems to learn a policy too fast to have a grasp of the environment (policy collapse), or it shows no learning at all. The balance between actor and critic could be an important factor that does not work for the particular setup presented. Additionally, the balance in the learning rate of the shared layers compared to the actor and critic heads could also be a factor in the problem. The reservoir computing experiment at evaluation was observed to dodge obstacles such as pipes. This is done without adapting the shared layers responsible for visual pattern recognition. This shows that the two heads were able to select action without adapting the patterns in the CNN's to the environment. A question that arises from these observations is whether the learning rate of the visual patterns in the data is too slow or too fast compared to the learning rate of the two heads of the actor-critic algorithm. Li et al. (2018) states that shared architectures for actor-critics perform better than separate architectures. However, when the imbalance between the learning rates of the components is a problem, it is easier to tune two separate networks.

5.1 Future Research

The experimental setup used has not utilized awareness of obstacles as a feature. Liao et al. (2012) uses a feature that gives the agent more information about its surroundings and the proximity of game elements. An agent that shows no learning could potentially benefit from more information regarding its surroundings.

Utilizing a stabler RL algorithm to do experiments on backup policies can give more insight into its influence during the learning phase. Furthermore, backup policies could be implemented in different ways. An experiment about implementing it as a feature in the network could be interesting. During this project, the backup policy was trained during the training of the normal agent. It

would be interesting to see what the effects of a pre-trained safety policy would have on the training of a normal policy.

The reward function of the backup policy showed different results. Punishing the proximity to a threat seems to have more potential compared to rewards that reward maximizing distance to a threat. Currently, no advanced methods are utilized to reward the agent for successfully evading a threat. However, generating an advanced safety reward system could be more effective than the proposed methods.

6 Conclusion

This study focused on the effect of a backup policy takeover to promote safety during training using an actor-critic algorithm. Results show no benefits of training a safety policy that rewards keeping distance from threats during the training of a normal policy in Super Mario Bros. There are hints in the observed training data that punishing proximity to threats using an alternate reward function could improve safety during training, but further steps are required to provide closure. The one-step actor-critic algorithm is not well suited to use as a base for experiments in the Super Mario Bros environment. The unstable learning that results in policy collapse makes it impossible to provide reliable data for exploring safety options during training. Furthermore, often before policy collapse or when policy collapse was not an issue, no signs of learning were observed from the data. Further Research needs to be done on an agent that shows learning to attain conclusive results on the effect of backup policies during the learning phase.

References

Ugur Doga Sezgin Jan Jones Adam Dingle, Jakub Gemrot. Github - medovina/marioai: Super mario implementation for experimenting with ai algorithms in java. `https://github.com/medovina/MarioAI`, 2012. (Accessed on 03/19/2024).

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. URL `http://arxiv.org/abs/1606.01540`. cite arxiv:1606.01540.

Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. ISSN 1935-8237, 1935-8245. doi:10.1561/2200000071. URL `http://arxiv.org/abs/1811.12560`. arXiv:1811.12560 [cs, stat].

Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1): 1437–1480, 2015.

Muzhe Guo, Feixu Yu, Tian Lan, and Fang Jin. Advantage Actor-Critic with Reasoner: Explaining the Agent's Behavior from an Exploratory Perspective, September 2023. URL `http://arxiv.org/abs/2309.04707`. arXiv:2309.04707 [cs].

Alexander Hans, Daniel Schneegaß, Anton Maximilian Schäfer, and Steffen Udluft. Safe exploration for reinforcement learning. In *ESANN*, pages 143–148, 2008.

Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. URL `https://github.com/Kautenja/gym-super-mario-bros`.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

Shangda Li, Selina Bing, and Steven Yang. Distributional advantage actor-critic, 2018.

Yizheng Liao, Kun Yi, and Zhe Yang. CS229 Final Report Reinforcement Learning to Play Mario. 2012.

Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer science review*, 3(3): 127–149, 2009.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *CoRR*, abs/1709.06009, 2017. URL `http://arxiv.org/abs/1709.06009`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

NesMaps.com. Super mario brothers - world 1-1 nintendo nes map. `https://nesmaps.com/maps/SuperMarioBrothers/`

SuperMarioBrosWorld1-1Map.html. (Accessed on 02/15/2024).

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A Survey of Deep Reinforcement Learning in Video Games, December 2019. URL http://arxiv.org/abs/1912.10944. arXiv:1912.10944 [cs].

Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2020. ISBN 978-0-262-03924-6.

Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. The Computational Limits of Deep Learning, July 2022. URL http://arxiv.org/abs/2007.05558. arXiv:2007.05558 [cs, stat].

A Appendix

Figure A.1: The actor loss compared for the different agent configurations.

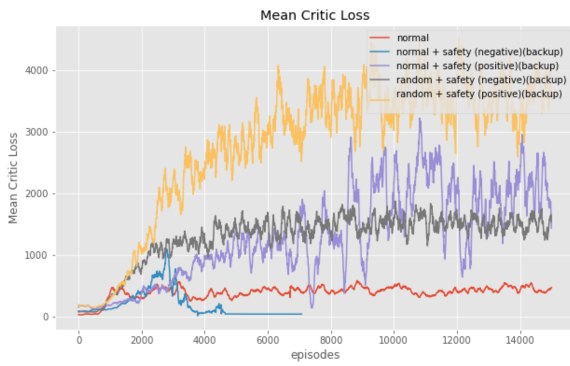

Figure A.2: The critic loss compared for the different agent configurations.

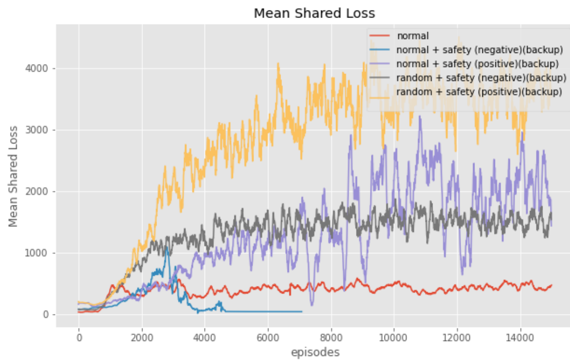

Figure A.3: The shared loss compared for the different agent configurations.