DEEP REINFORCEMENT LEARNING: GENERALISATION

Henriette Becker Kristiansen (s190192), Sebastian Larsson (s190619)

Technical University of Denmark

ABSTRACT

In this work we implement Deep Reinforcement Learning models based on the Proximal Policy Optimisation Algorithm using the Nature encoder and the Impala encoder. We propose a change to the reward system that penalises ending a game and empirically demonstrates that it results in longer games with a higher average score per game. The source code is available at

`https://github.com/SebastianLarssonDTU/02456-Reinforcement-Learning-Project`

1. INTRODUCTION

Generalisation, i.e. the ability to improve skills rather than memorising specific tasks, is a big challenge in deep reinforcement learning. In this study we look into the challenge of generalisation by attempting to teach an agent to play a simple arcade shooter game, StarPilot, from the Procgen suite of games [1]. As seen in [1], we explore models based on the the Proximal Policy Gradients [2] method, with encoders based on the encoders seen in [3] and [4]. To further improve the performance of this model, we propose a modification to the reward system that encourages self-preservation. With the best performing architecture and reward modification combined, we see a significant improvement to the performance. Additionally, the final model outperforms a small test-group of humans playing the game.

2. BACKGROUND

Reinforcement learning (RL) is an area of machine learning in which an agent can learn psudo-optimal action policies in a given environment through interactions. Deep learning (DL) is an area of machine learning that makes it possible to find tendencies in complicated data, using deep neural networks. A special case of DL are convolutional neural networks (CNN), which are state-of-the-art when it comes to image processing.

Deep reinforcement learning (DRL) is the combination of using deep learning to find tendencies in complicated data, and then running the output through a reinforcement learning model that can then learn a pseudo optimal policy. The use of DL in addition to RL makes it possible for the agent to learn certain properties about the environment, that it would not have been able to otherwise. DRL is especially useful when the agent does not have access to all the information about the environment, like is the case with computer games, where the agent or player does not know how the environment works, but can only observe a state.

Our agent is based upon the principle of a *stochastic policy network*, a neural network that given the state of the game will output a probability distribution over the possible action in the current state. When training the policy network, we sample from this distribution and execute the sampled action. In the classical case of supervised learning we would immediately know if this was the *right* action and could optimise the network accordingly to make the correct action more probable. But since this is unsupervised learning, we will have to wait and see if the sequence of actions we took had a *good* outcome. The *Policy Gradients solution* tackles this problem by sampling until a reward is achieved and then backpropagating to compute the gradients in a way that encourages the good outcomes and discourages the bad.

We will implement our agents based on the Proximal Policy Optimisation (PPO) algorithm (see more in section 3.2). The PPO algorithm is trying to combine the best of two worlds. It is based on the research done on Trust Region Policy Optimisation [5] (which have shown good data efficiency and reliable performance), while keeping the computation fairly simple (using only first-order optimisation) like seen in vanilla Policy Gradients.

The Proximal Policy Optimisation (PPO) algorithm we will be using is different from the vanilla Policy Gradients method in a couple of different ways. Our policy is updated after a *fixed number of timesteps*, and not once pr. reward or episode end. *The objective function is clipped* to ensure the policy updates are not too big.This will ensure that a noisy update does not result in a large policy update that might undo all the small steps the policy had taken in the right direction. This also allows us to *update with K epochs* without destroying the policy, resulting in better data efficiency. I.e. we learn more from each sample.

3. METHODOLOGY

3.1. The Procgen Benchmark

We are using the Procgen Benchmark to measure how good our model is at generalisation. The Procgen Benchmark [1] is an arcade game environment that has been specifically made for doing research in Reinforcement Learning. Procgen has 16 different arcade like games with procedurally generated levels. It is designed to help benchmark how well RL models generalise. The game rules, the action space, and the possible rewards, change from game to game in the suite. To train and test our models, we use only the game called StarPilot, in which the agent has to shoot down other space ships while avoiding all obstacles. If the agent is shot down or hits an obstacle, the agent dies, and the game resets.

3.2. Proximal Policy Optimisation Algorithm

In the Policy Gradients method we work with the gradient estimator, $\hat{g}$, given by

$$\hat{g} = \hat{\mathbb{E}}\left[\nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right) R_t\right]$$

Where π_θ is the stochastic policy network. a_t, s_t are the action and state at time t and R_t is the discounted future return from time t. i.e. given a series of T timesteps we have

$$R_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$$

where $\gamma \in [0, 1]$ is the discount rate and r_t is the reward obtained at time t.

3.2.1. Advantage function

We will be working with a fixed length trajectory version of the PPO-algorithm, which means that it samples for a fixed number of timesteps, independently of the actual episode length, and then updates the policy accordingly. For this reason we need to replace the future reward with an advantage estimator, $\hat{A}_t$, that only relies on the current sequence of timesteps. We use the generalised advantage estimator seen in [2], with decay parameter λ.

$$\hat{A}_t = \sum_{k=0}^{T-t+1} (\gamma\lambda)^k \delta_{t+k}, \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

where $V(s)$ is the state value function, i.e. the baseline estimate of how well we expect to perform when following the current policy from state s. $V(s)$ will be updated frequently by minimising the squared error between the current value estimate in state s and the actual reward seen during training.

3.2.2. Clipped Surrogate Objective

As proposed in [2] we use the clipped objective given by

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}\left[\min\left(q_t(\theta)\hat{A}_t, \text{clip}\left(q_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$$

where $q_t(\theta)$ is the probability ratio $q_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta old}(a_t \mid s_t)}$. I.e. $q_t(\theta)$ measures how likely the action a_t is under the new policy compared to how likely it was under the old policy. The hyper parameter ϵ decides how big policy updates we allow. Clipping is defined as follows

$$\text{clip}\left(q_t(\theta), 1-\epsilon, 1+\epsilon\right) = \begin{cases} 1+\epsilon & \text{if } q_t(\theta) > 1+\epsilon \\ 1-\epsilon & \text{if } q_t(\theta) < 1-\epsilon \\ q_t(\theta) & \text{else} \end{cases}$$

3.2.3. Final Objective

The final objective, taken from [2], is a combination of the clipped surrogate objective, $L_t^{CLIP}(\theta)$, a value function error term, L_t^{VF}, and an entropy bonus, $S\left[\pi_\theta\right](s_t)$.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t)\right]$$

where c_1, c_2 are hyperparameters. This is the objective used to update the policy.

The entropy bonus helps with balancing the exploration-exploitation dilemma - giving the model motivation to explore new actions instead of exploiting the perceived best course of action. The higher the coefficient c_2 is, the more likely the model is to explore.

3.2.4. The algorithm

The actual algorithm, as seen in [2], can be seen below.

Algorithm 1: Fixed length trajectory PPO algo.

for iteration = 1,2, … **do**
  for actor = 1,2 …, N **do**
    Run policy $\pi_{\theta old}$ in for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end
  Optimise surrogate L wrt. θ, with K epochs and minibatch size $M < NT$
  $\theta_{old} \leftarrow \theta$
end

3.3. Nature and Impala architecture

We decided to test two different encoders. The first, the nature encoder seen on figure 1, is based on the Convolutional Neural Network seen in [4]. The encoder is a fairly simple neural

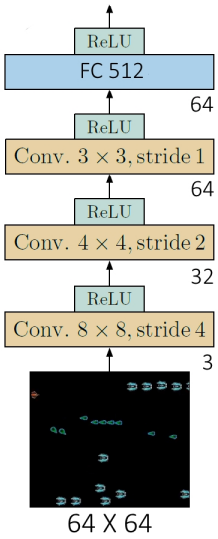

Fig. 1. Nature encoder architecture.

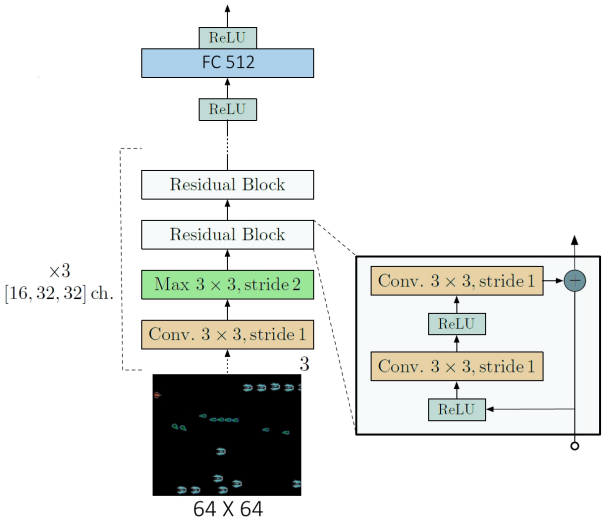

Fig. 2. Impala encoder architecture.

network with 3 hidden CNN layers and one fully connected layer.

The second encoder we are considering, the IMPALA encoder seen on figure 2, is based on the encoder seen in [3]. This encoder is more complicated, with 18 hidden layers: 15 convolutional, 3 max pooling layers and a fully connected layer.

3.4. Further modifications

Like seen in [6] we introduced clipping on the value function error term, V_t^{VF}, in a similar way as the clipping on the surrogate objective.

We add a *Death Penalty*, i.e. we add a fixed negative reward on game end.

3.5. Training and test set up

The chosen hyperparameters for our experiments can be seen in table 1.

The training is done continually on 256 steps, meaning that if the agent dies, the game simply resets, and in the next timestep, the agent has started on a new game. In the standard set up (without death penalty), the training reward is the cumulative normalised reward of all the shot down enemies during the 256 steps. Whether or not the agent dies, does not directly influence the reward.

When evaluating the agent the test score is calculated for whole episodes, and not for a fixed length like training. i.e. we calculate the average score per level.

When evaluating, we let the model play on 10 or 50 levels that it has never seen before, dependent on whether it was trained on 10 levels or more.

Hyperparameter	**Value**
Discount (γ)	0.99
GAE parameter (λ)	0.95
# of timesteps pr rollout	256
Batch size (Nature and modified Impala)	512
Batch size (Original Impala)	32
Value function coefficient (c_1) (Nature)	1.0
Value function coefficient (c_1) (Impala)	0.5
Entropy bonus (c_2)	0.01
PPO clip range	0.2
Reward Normalization	Yes
# of Workers	1
# of environments per worker	32
Total timesteps	8M
LSTM	No
Frame Stack	No
Optimizer	Adam
Learning rate (α) (Nature)	0.00025
Learning rate (α) (Impala)	0.0006

Table 1. The choice of hyperparameters for the models

Throughout training and testing we use the normalised reward, with the exception of when we compare with the human baseline, where we use the unnormalised reward.

4. RESULTS

For training and testing, we use the StarPilot game environment. We only train and test on easy difficulty and without any background. We train the models for 8M timesteps.

As can be seen on figure 3, the training performance follows the same trend whatever model we train.

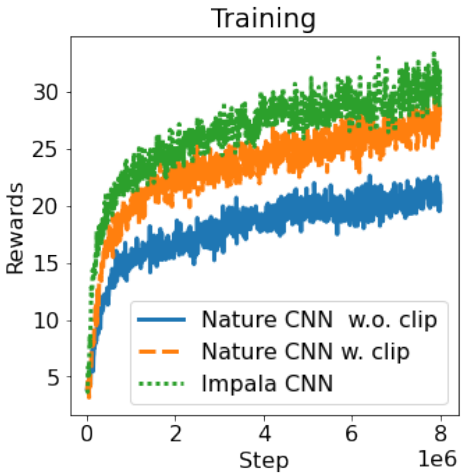

Fig. 3. For illustration purposes only a few models have been chosen out to be shown on the plot, but all the models follow this same trend during training.

On figure 4 we evaluate the Nature encoder against the Impala encoder trained on 10 levels. We test both architectures with and without value clipping. Impala is tested with batch sizes inspired by respectively [3] and [1]. It can be seen that both Impala and Nature performs significantly better with value clipping and Impala performs best with the batch size of 512. The Impala architecture performs better than the Nature architecture.

On figure 5 we test the effect of adding a death penalty to the Impala architecture (with value clipping and batch size 512). A death penalty of 1 seems to worsen the performance, but both 3, 5 and 7 leads to a better performance.

On figure 6 we test how the different architectures perform when trained on more levels. It is clear that the more levels the models are trained on, the better they generalise to levels they have never seen before.

Figure 7 shows the different architectures, trained on 200 levels, for 20 million steps.

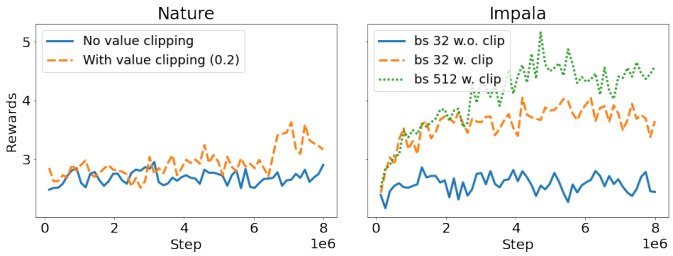

Fig. 4. Test performance of respectively Nature (Left) and Impala (right) with and without value clipping (clip). Impala further tested with different batch sizes (bs).

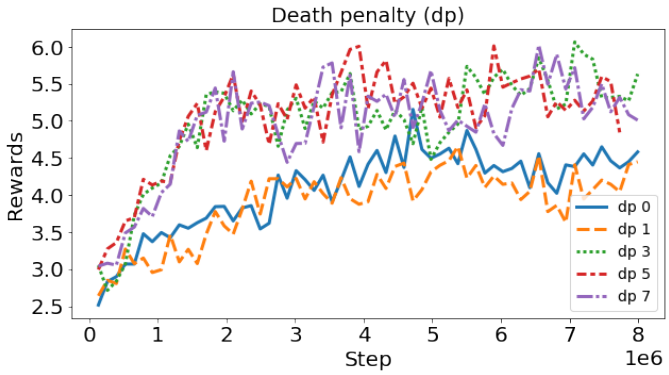

Fig. 5. Impala with value clipping, trained with different death penalties (dp).

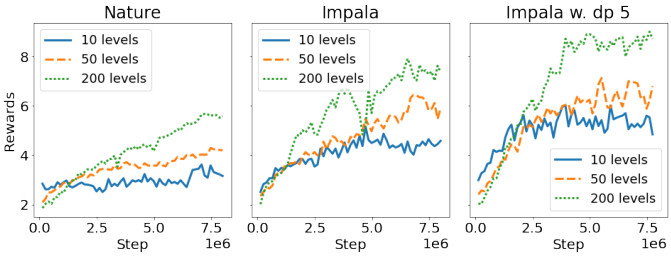

Fig. 6. Test performance of respectively Nature (left), Impala without death penalty (middle) and Impala with death penalty 5 (right) trained on 10, 50 and 200 levels.

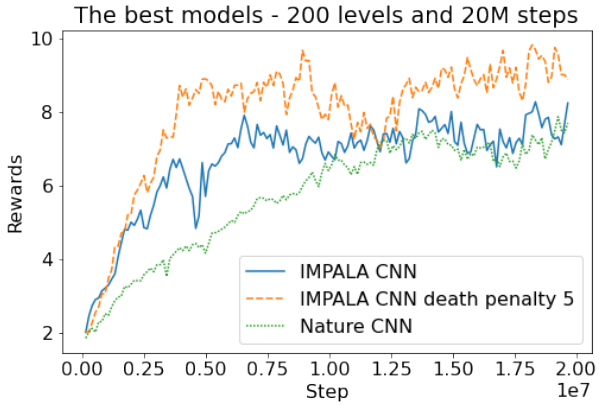

Fig. 7. The three best models trained with 200 levels have been trained on more steps to see if they perform equally well when trained on enough steps.

We compare the performance of the best models with three human test subjects who are used to playing computer games. For the sanity of the human test subjects, they were only asked to play for around 10-20 minutes. The last 20 levels from this session was used. As comparison: the models were trained on Colab for more than 12 hours each (including

time to evaluate during training). On table 2 we see that the models significantly outperforms the human test subjects.

Test human 1	Test human 2	Test human 3
14.15	14.55	18.1
Nature CNN	IMPALA w.o. dp	IMPALA w. dp. 5
22.14	27.16	34

Table 2. Average unnormalised reward over 20 levels. All of the models are trained on 200 levels.

5. DISCUSSION

Why is the training score so much higher than the test score? Firstly, the levels that the models are training on, they have seen before, partly allowing it to memorise specific solutions to these levels, while the levels that they are evaluated on afterwards, are new levels they have never seen before. Hence the evaluation performance is a measure of how well the models generalise. Secondly, the training performance is measured as the aggregated reward over 256 steps, while the evaluation performance is the reward from only one game, like mentioned in section 3.5. This means that the two numbers can not inherently be compared fairly. In general, the agents die considerably before using all 256 steps. However, a better training performance usually leads to a better evaluation performance.

How do the encoders compare to each other? We chose the Nature and Impala encoders as baselines so we could compare our initial result to the results seen in [1]. But we quickly chose to focus on the Impala encoder since it is widely used and known for being a good encoder for image processing with more expressive power than the Nature encoder. So as expected, we see that the Impala encoder learns faster than the Nature, but that it does eventually seem to catch up, as can be seen on figure 7.

Why did we consider death penalty? Watching some example videos of the models trained, we saw a trend: The agents seems to learn to play a lot of short games with few points instead of longer games with more points. Therefore we proposed a change to the credit assignment by adding a penalty for ending a game. By modifying the reward system we were hoping to encourage the agent to stay alive and as a result getting longer games with a higher point total.

What is the effect of death penalty?

As can be seen on figure 5 and figure 6, the models with implemented death penalty performs significantly better than the other models. When generating example videos of the different agents playing, we can see that the death penalty have resulted in the wanted behaviour: the agent now has self-preservation as a priority. For illustration purposes, some distinct examples have been uploaded to the git repository and can be found in the Videos folder. In these examples it can be seen how the agent without death penalty ignores the shot that kills it, instead it is prioritising aligning itself with, and shooting, the blob of enemies in front of it. This way it accumulated a lot of points in a short time frame. The agent trained with death penalty instead actively avoids the shots coming at it, sacrificing the possibility of shooting down the last ship in the blob, and instead moves up.

In this case we really see the impact of how credit assignment influence the performance of the agents. The rewards the agent gets while it trains is the only way it can improve and *learn a task*. So how we choose to reward/punish is fundamental for the agent to learn. In other words: there is a lot of performance to be achieved by spending some time reinventing/reconsidering the task-specific credit assignment.

Will the agent eventually learn self-preservation without death penalty? We speculate that the agent might eventually learn self-preservation on its own. By exploring enough games and realising it is more time efficient to stay alive and kill the enemies on screen rather than dying repeatedly and waiting for enemies to move onto the screen (they all spawn off-screen in the beginning of a level). But we have not been able to test this hypothesis due to time constraints. Even for the models that have trained for the most timesteps (see figure 7) the models, with and without death penalty, have not converged. So we can not say whether they eventually converge or if death penalty will maintain its lead.

Why did we choose these hyperparameters? The model hyperparameters (seen in table 1) are inspired by [1] and [3]. Assuming that the hyperparameters in [1] are pseudo optimal for using the Procgen Benchmark with the PPO algorithm, we have not done any further hyperparameter tuning. However, the hyperparameters specified in [3], have been optimised for the Impala algorithm on a different dataset. So we test the hyperparameters both as seen in [3] and modified to have the same batch size as seen in [1].

What effect has the computational constraints had on our choices? We have trained all models using the free compute power at Google Colab. To be able to train a reasonable amount of models we only train and test on easy difficulty and without any background. Through testing, we found that 8M training steps resulted in a reasonable performance and could be done within the time constraints on Colab. Comparing to the figures in [1], where they train for 25M timesteps, we can see that the same trends are visible already after 8M timesteps.

How well does Death Penalty generalise to other games? It can be argued that any modification to the reward system is very game dependant and will not help generalise models when training on different games. But in the specific setting of computer games we speculate that a big part of them will benefit from a penalty on game end, since many games will accumulate points for as long as the game is running. Other games might be indifferent to this change (fixed length games or never ending games). Some games might

even see a decrease in performance (Games where the goal is to finish fast and the death penalty is disproportionate to the game reward itself).

What further improvement would we pursue given more time? One pursuit would be to add data augmentation as seen in [7], where they show that it can also improve performance further. For StarPilot they have shown that *cutout*, a method where small patches of the frames are "cutout" and replaced with black or coloured pixels, seems to give a significant improvement. Another suit of action would be to implement frame stacking. The notion of movement is not captured in a single frame, so conceptually the agent should learn to avoid obstacles better when it can get a sense of direction and speed from processing multiple frames at once. In [1] they specifically show that using frame stacking with the Impala encoder improves performance for the StarPilot game.

6. REFERENCES

[1] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman, "Leveraging procedural generation to benchmark reinforcement learning," *CoRR*, vol. abs/1912.01588, 2019.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[3] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu, "IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures," *CoRR*, vol. abs/1802.01561, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[5] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015.

[6] Nipun Wijerathne Varuna Jayasiri, "Labml: A library to organize machine learning experiments," https://labml.com/labml_nn/rl/ppo/, 2020.

[7] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas, "Reinforcement learning with augmented data," 2020.