Contents

I List of Figures

II Abbreviations

NN Neural Network
PPO Proximal Policy Optimisation
RL Reinforcement Learning
TRPO Trust Region Policy Optimization

1 Introduction

Although already being actively researched since the 1950s, reinforcement learning (RL) only gained popularity rather recently in 2013 with Mnih et al. [Mnih et al., 2013], demonstrating that reinforcement learning is able to win Atari games against the computer as well as partially against humans. The researchers of the company DeepMind achieved this by combining reinforcement learning with deep learning. In 2016, DeepMind managed to design a reinforcement learning model which exceeded human performance in the complex game of Go [Géron, 2019]. This paper proposes a deep reinforcement learning algorithm implementation for the game of Pong, using proximal policy optimization (PPO) and demonstrates its functionality and convergence. Furthermore, its reward development stability (or result stability) is investigated, and two different neural network (NN) implementation architectures are compared.

2 Pong and Gym

The video game Pong, introduced by Atari in 1972, is one of the earlier computer games and gained much popularity during the 1970s and 1980s. Similar to tennis, two players each have one paddle and try to play a ball in a way to bypass the opponents paddle, which results in a point for the party having last touched the ball [Atari, 1977]. It can be played by two humans or by one human playing against the computer.

With the advent of reinforcement learning, it became a training and comparison environment for reinforcement learning algorithms, which play games versus the computer. For Python, the library Gym offers an Atari environment, which includes Pong and allows for an orderly and pythonic interaction of the reinforcement learning algorithms and the game. It returns the current frame of the game as an RGB image, which can then be further processed [Brockman et al., 2022]. In Pong, a game - or *episode* in the context of reinforcement learning - has in between 21 and 40 *rounds* and ends with one party winning when achieving 21 points.

Pong comes in several versions in Gym, some being deterministic while others randomly skip frames to emulate a less globally observable environment [Brockman et al., 2022]. This paper uses *PongDeterministic-v4*.

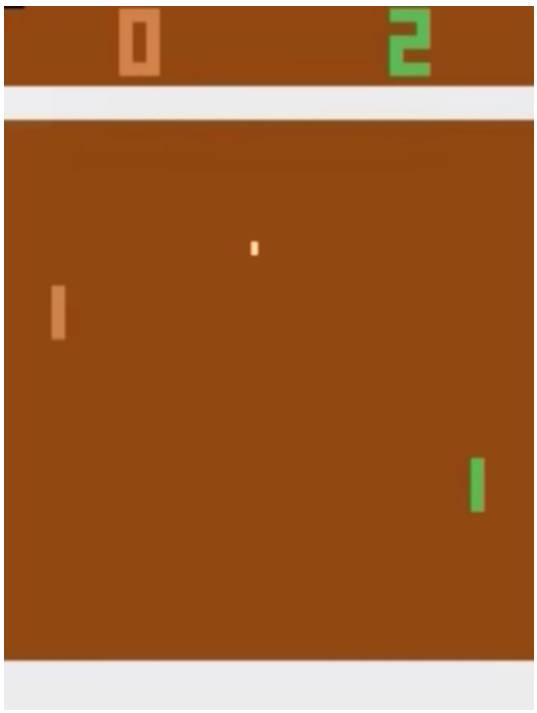

Figure 2.1: Example screen of Pong as depicted in Gym [Brockman et al., 2022]

3 Reinforcement Learning

Reinforcement learning is a field of machine learning in which a model, or agent, is trained to learn mapping situations to actions in a way that optimizes a numerical indicator of reward [Sutton and Barto, 2018]. Compared to supervised learning, RL is considered harder as the reward is generally sparse, noisy and delayed, while states are highly correlated [Mnih et al., 2013]. Deep RL is consequently actively researched and offers a vast landscape of different algorithms and mathematical approaches. An overview is out of scope for this paper, the reader is referred to [Sutton and Barto, 2018, Arulkumaran et al., 2017, Zhang et al., 2021] for further information.

Generally, the algorithms vastly differ in effectiveness, efficiency and implementability. Schulman et al. [Schulman et al., 2017] state that improvement potential for scalable algorithms is visible. In the field of deep RL, the most mentioned and best performing algorithms work by policy gradient optimization, except for Q-learning. As, however, Q-learning is poorly understood and often fails on simple problems, this paper focuses on gradient based methods [Schulman et al., 2017].

3.1 Policy Gradient Based Reinforcement Learning Algorithms

Generally, the idea of policy gradient optimisation is to maximize the estimator of the policy gradient, which returns the optimal policy as a result. The most common estimator is defined as

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta log \pi_\theta(a_t | s_t) \hat{A}_t \right] \tag{3.1}$$

where π_θ is a stochastic policy of action a_t given state s_t at time t, and $\hat{A}_t$ being the estimate of the advantage function at time t [Schulman et al., 2017]. π_θ takes the observed states from the environment and outputs a distributions of actions to take.

To calculate $\hat{A}_t$, the discounted sum of rewards and a baseline estimate, or value function, are needed.

$$\hat{A}_t = Discounted\ Rewards\ -\ Baseline \tag{3.2}$$

The discounted sum of rewards is described by

$$\sum_{k=0}^{\infty} \gamma^k r_t + k \tag{3.3}$$

with r_t as the reward and γ as the discount factor, incorporating that the algorithm should lay higher weight on timely closer achieved rewards. Generally, the returns are deterministic in our advantage function as they are already returned by the environment the moment we calculate the advantage function.
The baseline or value function provides an estimate of the discounted rewards from this point onwards, using a neural network which is frequently updated during the training process. This neural network is also called the Critic.
$\hat{A}_t$ can consequently be interpreted as the improvement our current action brought compared to the expectation of the normal event to happen in the current state.

Repeated training of the critic using the same batch of data leads to a noisy critic and too large policy jumps. This is problematic, as the algorithm might only slowly converge to an optimal solution, or find none at all (see appendix A.1). Additionally, vanilla policy gradients are not considered data efficient [Schulman et al., 2017]. Schulman et al. [Schulman et al., 2015] proposed Trust Region Policy Optimization (TRPO) to prevent those problems. In their paper, they slightly modify 3.1 by changing

$$log\pi_\theta(a_t|s_t)$$

to

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3.4}$$

and adding a KL-constraint to the objective function of the optimization. The overal TRPO objective function is then

$$\max_\theta\ \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \tag{3.5}$$

$$subject\ to\ \hat{\mathbb{E}}_t[KL[\pi_{\theta old}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \sigma \tag{3.6}$$

These two modifications prevent excessive policy improvement steps which might lead to jumping and inconsistent results. However, adding the Kullback-Leibler Divergence as a constraint increases the computational overhead of the optimization [Schulman et al., 2017].

3.2 PPO as Algorithm of Choice

To reduce the cost of calculation and facilitate the mathematical model, Schulman et al. [Schulman et al., 2017] proposed PPO in 2017, incorporating the improvements of TRPO while reducing complexity by the usage of loss clipping.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t] \tag{3.7}$$

Here, $r_t(\theta)\hat{A}_t$ is the normal policy gradient objective, which is repeated in the second minimization objective $clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)$, with the difference, that $r_t(\theta)$ can only take values within the supplied range. This ϵ is used in the aforementioned loss clipping, which prevents that the policy changes to much in one step. It usually takes the value 0.2, as also proposed by Schulman et. al. [Schulman et al., 2017].

The overall objective, which is maximized every iteration is then given by:

$$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\,[\pi_\theta](s_t)] \tag{3.8}$$

where $c_1 L_t^{VF}(\theta)$ updates the baseline network and $c_2 S\,[\pi_\theta](s_t)]$ is an entropy term, automatically ensuring that the agent does enough exploration, if needed. The reader is referred to [Schulman et al., 2017] for further information to those terms.

In the PPO-introducing paper, Schulman et al. [Schulman et al., 2017] found that PPO not only reduces complexity, but also performs better than TRPO when compared. However, Engstrom et al. [Engstrom et al., 2020] contested this and attributed much of the performance improvement to code level improvements. Still, as its efficiency improvement remains and as it is one of the best-performing algorithms, it is chosen for this paper.

4 Model setup

Generally, the model is written in Python and uses Keras for the neural networks. Within the overall code structure, we first define `NN_Model()`, which is used in our main class `PPOAgent`.
When starting the python script, the PPO model adheres to the following simplified procedure to learn: For each game in episodes:

1. Reset Environment and Necessary Variables
2. While Game (or Episode) Not Done
    a) Predict Action Using the Actor Network

b) Do Step: Supply Action to Environment and Retrieve Next Image, Reward and Done Information

c) Process Image From Gym

3. Compute Discounted Rewards
4. Predict Values Using the Critic Network
5. Compute the Advantages
6. Fit Actor and Critic Networks
7. Save Actor Model If Improved
8. Increment Game (Start Next Game)

4.1 Function `NN_Model()`

The function `NN_Model()` defines the general model of our actor and critic. As PPO is a deep reinforcement learning algorithm, the model includes neural networks, which are defined and compiled for both actor and critic in this function. For this, Keras is used. Different architectures may be used here, some can be found in the full code example.

```
def NN_Model(input_shape, action_space, lr):
    X_input = Input(input_shape)
    X = Flatten(input_shape=input_shape)(X_input)
    X = Dense(512, activation="elu", kernel_initializer='he_uniform')(X)
    # further NN architectures in full code
    action = Dense(action_space, activation="softmax",
     ↪ kernel_initializer='he_uniform')(X)

    if shared_nn:
        value = Dense(1, kernel_initializer='he_uniform')(X)
    else:
        Z = Flatten(input_shape=input_shape)(X_input)
        Z = Dense(512, activation="elu", kernel_initializer='he_uniform')(Z)
        value = Dense(1, kernel_initializer='he_uniform')(Z)

    def ppo_loss(y_true, y_pred):
        # Defined in https://arxiv.org/abs/1707.06347
        advantages, prediction_picks, actions = y_true[:, :1], y_true[:,
         ↪ 1:1+action_space], y_true[:, 1+action_space:]
        LOSS_CLIPPING = lossclipping
```

```
        ENTROPY_LOSS = 5e-3

        prob = y_pred * actions
        old_prob = actions * prediction_picks
        r = prob/(old_prob + 1e-10)
        p1 = r * advantages
        p2 = K.clip(r, min_value=1 - LOSS_CLIPPING, max_value=1 +
        ↪  LOSS_CLIPPING) * advantages
        loss =  -K.mean(K.minimum(p1, p2) + ENTROPY_LOSS * -(prob *
        ↪  K.log(prob + 1e-10)))

        return loss

    Actor = Model(inputs = X_input, outputs = action)
    Actor.compile(loss=ppo_loss, optimizer=RMSprop(learning_rate=lr))
    Critic = Model(inputs = X_input, outputs = value)
    Critic.compile(loss='mse', optimizer=RMSprop(learning_rate=lr))

    return Actor, Critic
```

As described in chapter 3.1, PPO applies loss clipping for the actor. In this code, it is defined within the function `ppo_loss(y_true, y_pred)`. Being based on 3.7,

$$r_t(\theta)\hat{A}_t = \texttt{p1}$$

and

$$clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t = \texttt{p2}$$

using the hyperparameter `lossclipping`, which is set to 0.2 (not visible in this snippet), following the proposal in [Schulman et al., 2017]. $r_t(\theta)$ additionally features a very small denominator constant to prevent a division-by-zero error if `old_prob` = 0.

Huang et al. [Huang et al., 2022] find that PPO performs significantly better with completely separated networks for actor and critic. This will be tested in chapter 5.2. The standard version of the code features a shared network architecture.

4.2 Class `PPO_Agent`

The class `PPO_Agent` stores most of our functions and hyperparameters. It initializes the neural networks and the overall model in it's `__init__()` function.

```
def __init__(self, env_name):
    # Initialization
```

```
        # Environment and PPO parameters
        self.env_name = env_name #'PongDeterministic-v4'
        #self.env = gym.make(env_name, render_mode='human') # uncomment this to
         ↪  get a visual representation
        self.env = gym.make(env_name)
        self.action_size = self.env.action_space.n
        self.EPISODES, self.episode, self.max_average = eps, 0, -21.0 # specific
         ↪  for pong
        self.lock = Lock() # lock all to update parameters without other thread
         ↪  interruption
        self.lr = learningrate

        self.ROWS = 80
        self.COLS = 80
        self.REM_STEP = 4
        self.EPOCHS = 10 #Number of times the same observations are used to
         ↪  train the NNs

        # Instantiate plot memory
        self.scores, self.episodes, self.average = [], [], []

        # Create Actor-Critic network model
        self.Actor, self.Critic = NN_Model(input_shape=self.state_size,
         ↪  action_space = self.action_size, lr=self.lr)
```

`self.EPISODES` is the number of episodes the algorithm is trained on, while the lower case `self.episodes` represents the current episode and is in essence an incremental counter. `self.max_average` is a starting value for the maximum average reward, which is later used as an indicator whether the model improved compared to its previous form. `self.lr` represents the learning rate for the neural networks, which is set in `__main__`. This paper adheres to the usual setting of 0.0001. `self.REM_STEP` specifies how many past frames are incorporated for the training and `self.EPOCHS` defines the number of fits for the neural net for one experience: At 10, the neural networks are fitted for ten times with the available data. Also see Appendix A.1 for a short experiment on the EPISODE hyperparameter.
All further functions of `PPO_Agent` can be found in the GitHub repository.

5 Results and Analyses

In this chapter, the results of the PPO model are outlined and described. Where not differently indicated, the standard hyperparameters are

- a dense, one-layer neural network with 512 neurons and elu activation, which is shared for actor and critic
- loss clipping of 0.2, following Schulman et al's [Schulman et al., 2017] proposal
- a learning rate of 0.0001 for the neural networks
- 750 episodes.

This list is by far not exhaustive, all details can be found in the code repository on GitHub.

5.1 General Results

Generally, the actor learns the game in all runs and always manages to reliably and sustainably win the game. The fourth run is exemplary depicted in 5.1, all runs (with standard parameters) can be found in the next section. It should be noted that the fourth run is neither the best nor the worst run.

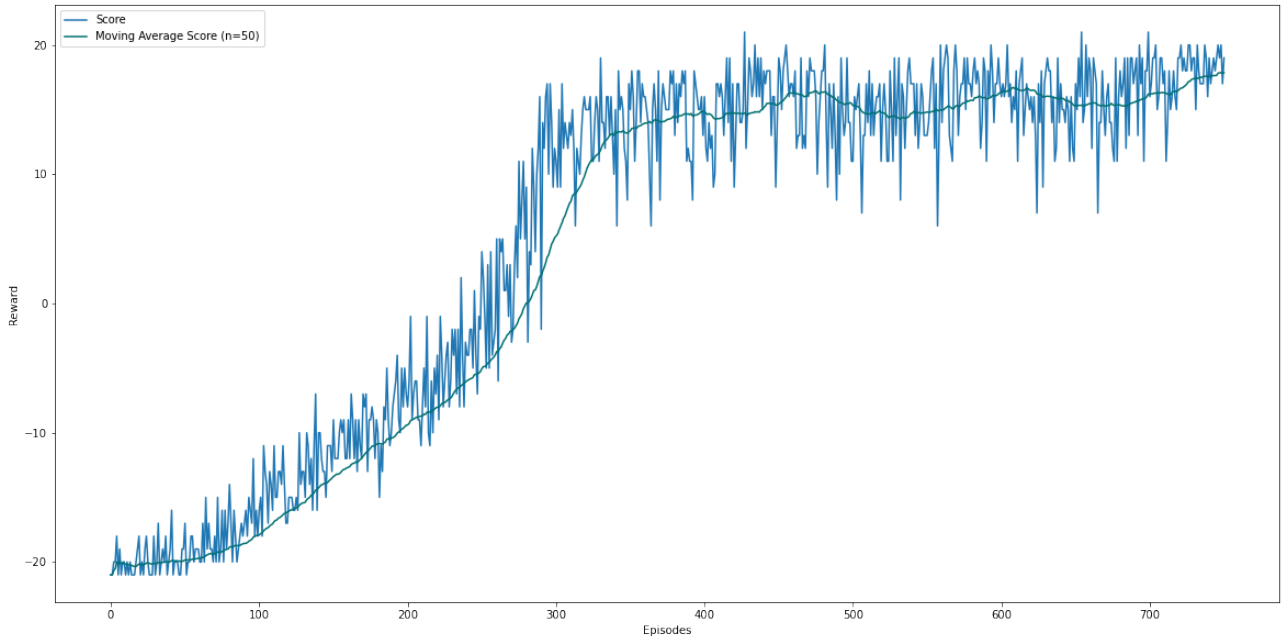

Figure 5.1: Reward Development over Episodes of Fourth Run

It can be observed that most of the improvement is achieved within the first 330 Episodes. Afterwards, only marginal and also not necessarily reliable improvement can be observed.
In between consecutive episodes, some variance can be observed. Although this can be quite significant and might include larger jumps, catastrophic forgetting - a common occurrence where neural networks forget their past experiences when seeing new data - can not be observed [McCloskey and Cohen, 1989]. As the loss is clipped, this observation confirms the expectation.

5.2 Result Stability in Reinforcement Learning

Reproducing experiments in reinforcement learning is considered notoriously hard, due to the richness in hyperparameters, and the sensitivity to code level optimisations [Henderson et al., 2018b, Huang et al., 2022, Tucker et al., 2018]. Henderson et al. [Henderson et al., 2018a] find that the diversity of metrics and missing significance testing of results in literature might lead to misleading depiction of results.

While this is a well-known fact in RL research, Henderson et al. [Henderson et al., 2018a] also show that RL algorithms have high variance across runs - an observation we also find for our PPO model. To properly report the results, we first display them and then comment on the variance in between runs. All results of the nine runs are displayed, following Henderson et al's [Henderson et al., 2018a] proposal of not only reporting the top n results of m runs, as popular papers often do [Mnih et al., 2016, Wu et al., 2017]. Additionally, results are often averaged over low counts of runs, lowering credibility [Wu et al., 2017, Gu et al., 2017]. To circumvent this, we display all runs. For readability, we do not include all individual results, but the moving average over 50 episodes. The first runs were only done until episode 450, however, they are still included as most of the results focus on the episode range below 450.

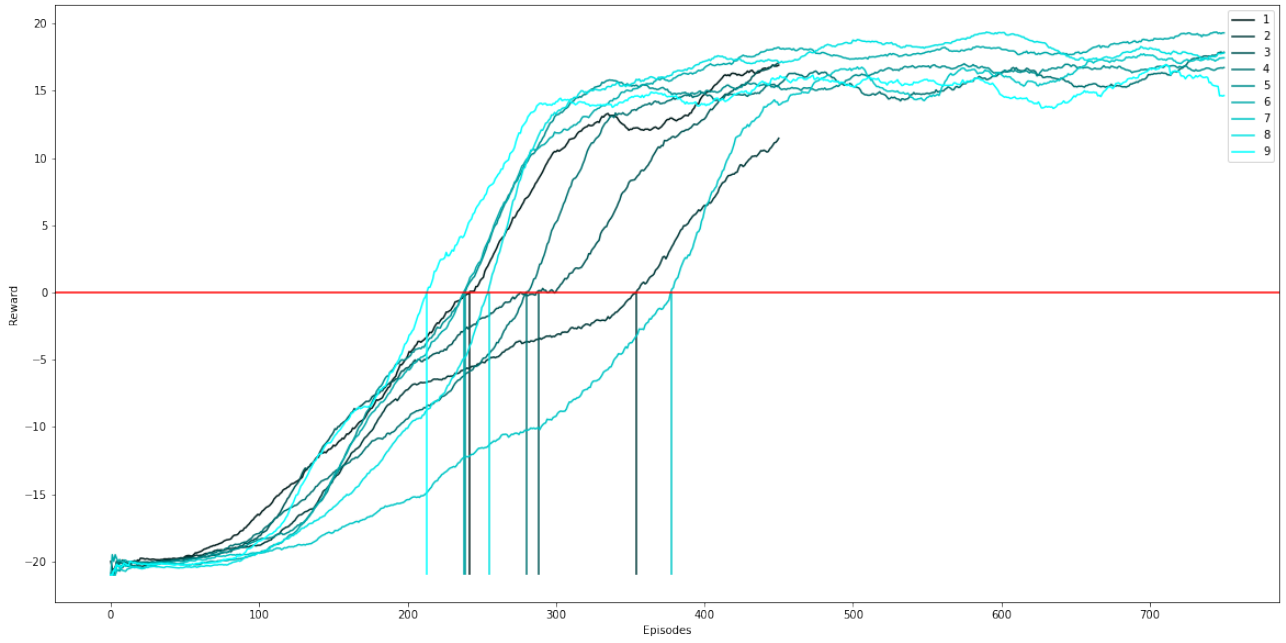

Figure 5.2: PPO Result Stability over 9 Runs and 750 Episodes

Generally, one can observe that no major and sustainable learning takes place during the first 100 Episodes. As of episode 120, the learning accelerates until a reward of 14-15 is reached. As of then, learning only progresses slowly, if improvement is visible at all.
Overall, a large variance can be observed in between the runs. While the agent learns really fast for some runs, as for example run 9, it is a quite slow learner for others, as for example run 7. The best algorithm wins after 213 episodes, the worst after 378 and on average after 276 episodes. This coincides with existing research of RL algorithms by Henderson et al. [Henderson et al., 2018a] for

different environments than Pong.

5.3 Improvement of Separate Neural Network Architecture Over Shared Neural Network Architecture

Huang et al. [Huang et al., 2022] found that one major implementation detail of PPO was the usage of seperate neural networks for actor and critic networks. They found that PPO algorithms using completely separate networks were vastly outperforming the standard design of shared network architectures for simpler environments after benchmarking the results for the environments CartPole, Acrobot and MountainCar [Huang et al., 2022].

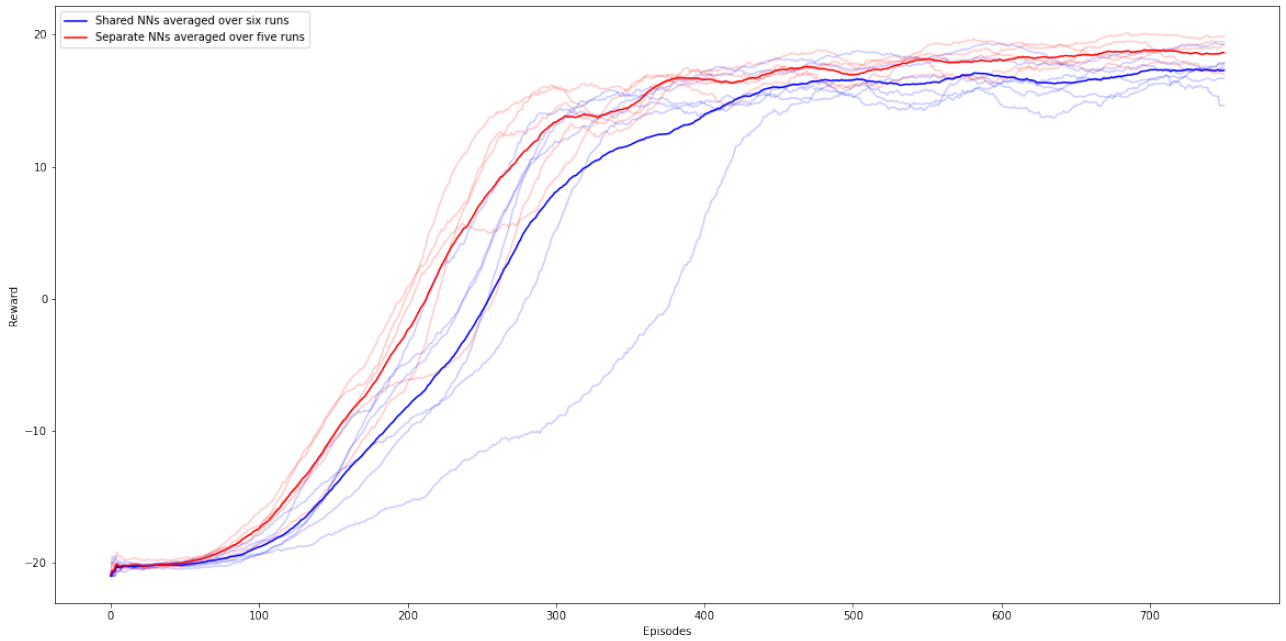

Figure 5.3: Reward over Episodes of Shared Neural Network Architecture versus Separate Neural Network Architecture for Actor and Critic.

Repeating these experiments for Atari's Pong (keeping all other hyperparameters fixed, including the neural network's), we find that improvement of separate network architecture over shared network architecture is visible, but less pronounced than for Huang et al's [Huang et al., 2022] experiments. Still, separate network architecture on average clearly outperforms shared network architecture at any point in time after the initial 50 episodes. Furthermore, it is visible that the separate network architecture approaches the global optimum of a reward of 21 better than the shared networks. It is also visible that the overall result stability seems to be better when using a separate network architecture as the variance in between the runs appears lower.
However it should be noted that these experiments were only comparing 5, respectively 6 training runs and the result might therefore also be a result of chance. Further repetitions of algorithm training might increase the confidence into these findings.

6 Conclusion

This paper introduced an implementation of the PPO algorithm for Pong and demonstrated its results, following clear and comprehensive documentation standards. It investigated the stability of the implementation and compared shared NN architecture to separate NN architecture, finding that separate NNs for actor and critic outperform shared NN architecture.

Further repetitions of the experiments might increase the confidence in the results. Also, separate versus shared NN architectures could be compared in further differing PPO setups and for differing hyperparameters or nn architectures (e.g., convolutional neural networks) to investigate whether separate NN architectures show better performance on a more global level.

Bibliography

[Arulkumaran et al., 2017] Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38.

[Atari, 1977] Atari, I. (1977). Video olympics. `https://atariage.com/manual_html_page.php?SoftwareLabelID=587`. Accessed: 2022-08-21.

[Brockman et al., 2022] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2022). Pong. `https://www.gymlibrary.ml/environments/atari/pong/`. Accessed: 2022-08-21.

[Engstrom et al., 2020] Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., and Madry, A. (2020). Implementation matters in deep policy gradients: A case study on ppo and trpo. *arXiv preprint arXiv:2005.12729*.

[Géron, 2019] Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.".

[Gu et al., 2017] Gu, S. S., Lillicrap, T., Turner, R. E., Ghahramani, Z., Schölkopf, B., and Levine, S. (2017). Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Henderson et al., 2018a] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018a). Deep reinforcement learning that matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

[Henderson et al., 2018b] Henderson, P., Romoff, J., and Pineau, J. (2018b). Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods. *CoRR*, abs/1810.02525.

[Huang et al., 2022] Huang, S., Dossa, R. F. J., Raffin, A., Kanervisto, A., and Wang, W. (2022). The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

[McCloskey and Cohen, 1989] McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA. PMLR.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[Schulman et al., 2015] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust region policy optimization. *CoRR*, abs/1502.05477.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

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

[Tucker et al., 2018] Tucker, G., Bhupatiraju, S., Gu, S., Turner, R., Ghahramani, Z., and Levine, S. (2018). The mirage of action-dependent baselines in reinforcement learning. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5015–5024. PMLR.

[Wu et al., 2017] Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Zhang et al., 2021] Zhang, K., Yang, Z., and Başar, T. (2021). Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, pages 321–384.

A Annexes

A.1 PPO Training for Varying Values for EPOCHS Parameter

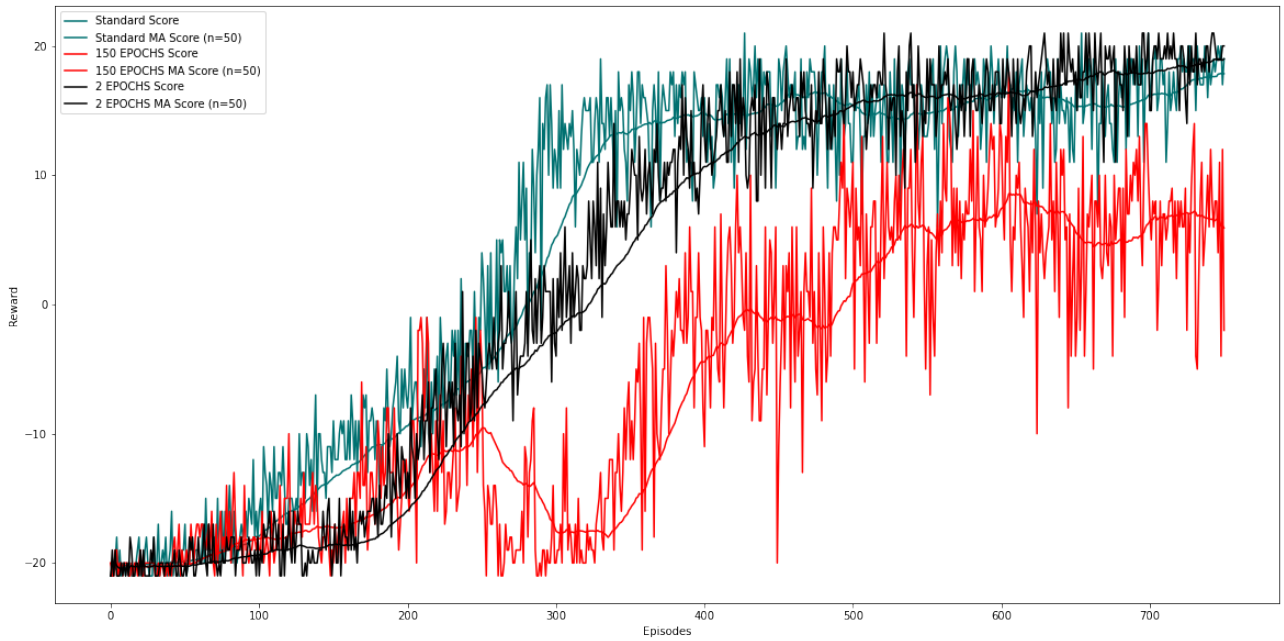

Figure A.1: PPO Reward Development for Varying Values of EPOCHS Parameter.

In this small trial, it can be observed that compared to the baseline EPOCHS=10, EPOCHS=150 is very unstable and by far less performant. It is to be assumed that this is caused by significant overfitting of the neural networks, being fitted 150 times with the same data for each episode. The strongly oscillating score is a further indicator for this.
EPOCHS=2 learns a bit slower than the base line algorithm, but performs better than the base line as of episode 600. This might be an indication that for late stage training, the baseline is also overfitting its neural networks.
However, it must be noted that these hypotheses need further investigation. As described, RL is unstable in its results, so repetitions of these experiments are needed to validate these conclusions.