Neural Networks as Reinforcement Learning Agents Winning Atari Pong

Johannes Kürsch, Jonathan Lennartz and Jan Nogga

Universität Bonn, Germany
kuersch@uni-bonn.de, jlen@uni-bonn.de, jan@nogga.eu

Abstract

We illustrate the topic of reinforcement learning with neural networks by exemplarily training a deep reinforcement learning agent to play Atari Pong in OpenAI Gym. We evaluate various training set-ups and showcase efficient implementations thereof in the PyTorch deep learning framework. Training is expedited by performing computationally expensive operations on a CUDA-enabled GPU.

1 Introduction

Merging the long-standing and flexible paradigm of reinforcement learning with the ability to construct knowledge directly from raw input achieved by deep neural networks has enabled a class of artificial agents capable of human-level performance on challenges diverse and dissimilar. For this reason deep reinforcement learning is a major topic of on-going research. Our goal is to render this field accessible by providing its foundations in theory and detailing our efforts in transferring them to tangible applications. In the interest of presenting a flexible framework and preserving comparability of our results to other work, we make use of OpenAI Gym.

2 OpenAI Gym

OpenAI Gym is a comparison toolkit for reinforcement learning algorithms and provides numerous curated environments to test these algorithms on [1]. The environments range from basic control tasks, such as balancing a pole on a cart in two dimensions, several Atari 2600 games like Breakout or Space Invaders, environments to learn imitating computations, for instance multi-digit addition, or advanced three-dimensional robot control tasks, including grasping and fetching.

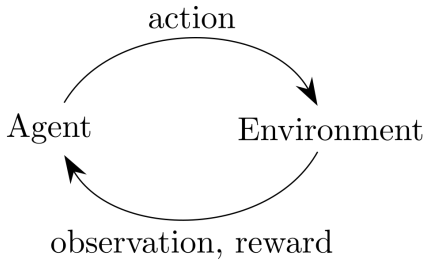

Figure 1: OpenAI Gym work flow [1]

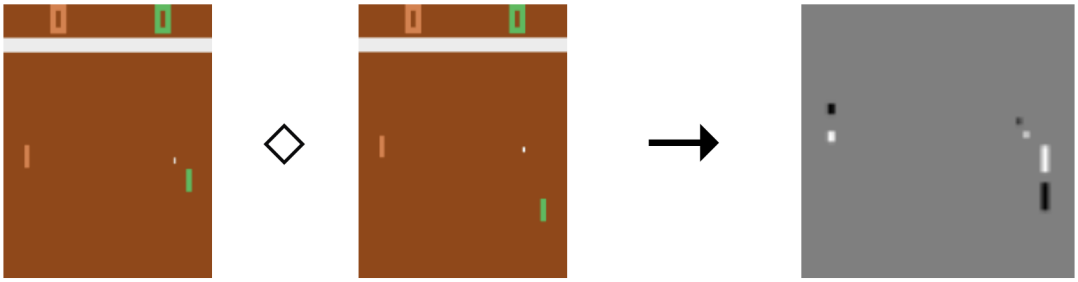

Figure 2: Preprocessing of Pong frames

2.1 Interface

OpenAI Gym provides an interface to Python with the *step* function at its core. Passing the chosen *action* to the *step* function returns an *observation* of the environments resulting *state*, an associated *reward* and further information, most notably whether the state is terminal, that is to say, whether the particular episode played in the environment is done. This process is shown in Figure 1.

2.2 The Pong Environment

In our project we worked with different versions of the Pong environment. Pong is a simple two-dimensional Atari game that simulates table tennis; the computer and the player control one paddle each and a point is scored if the opponent fails to return the ball. To win a match a player has to score 21 points.

Every observation of the environment consists of a frame with 210 by 160 RGB pixels. Before passing this data through our policy network, basic preprocessing is conducted. The upper 35 rows of pixels containing the score are truncated, as the reward signal from the environment obviates the necessity of displaying this information. Furthermore, the picture is downsampled by excluding every second pixel, resulting in an 80 by 80 pixel image. Subsequently, the image is binarized by zeroing the background and setting the values of paddles and ball to one. In a final step the difference of two consecutive frames is calculated to encode velocities and their directions. In this sense, the output of the preprocessor can be considered a full representation of the environments state. The preprocessing is illustrated in Figure 2. Note that this preprocessor and the architecture seperating model and solver of our code mimic an implementation in the MinPy documentation we were given as a starting point [2].

The Pong environments we worked with skip several frames of play between the input of the agents action and the observation which is returned. For these inaccessible states, the selected action is simply repeated. Here, we focus on the environment *PongDeterministic*, in which the number of frames skipped is always 4.

It should be stressed that the adversarial AI controlling the left paddle is the same in all versions. According to OpenAI, it significantly outperforms humans, winning with an average lead of 11 points [2].

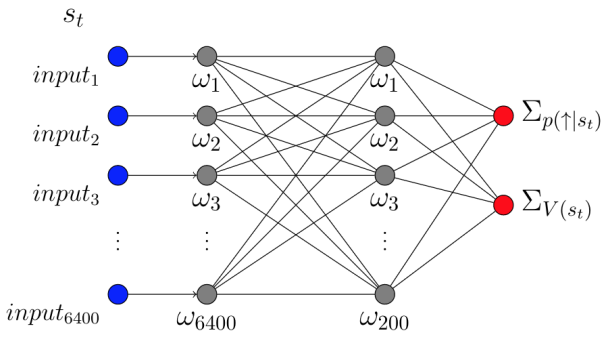

Figure 3: Fully connected policy network

3 Reinforcement Learning

Richard Sutton and Andrew Barto, two founding fathers of computational reinforcement learning, concisely describe it as "learning what to do - how to map situations to actions - so as to maximize a numerical reward signal" [7]. In more formal terms, our standard setting is an agent interacting with an environment over a number of discrete time steps. At each time step t the agent receives the environments state s_t, chooses its action a_t following a policy $\pi(s_t, a_t)$ and the environment provides s_{t+1} as well as the resulting reward r_t in turn. The goal is to maximize the expected accumulated discounted future rewards $\mathbb{E}\big[R_t\big]$ for each s_t. Specifically, the return R_t is defined by

$$R_t \coloneqq \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Here, $\gamma \in [0, 1]$ is a discount factor used to prioritize short versus long-term rewards. In our experiments, we used $\gamma = 0.99$.

4 Policy Networks

The preprocessed observations are passed through our policy networks, which respond with a probability of moving the paddle upwards and an estimate of the return which will follow this particular state. Essentially, we investigate two different network architectures. Our first attempts were centered around a fully connected network, depicted in Figure 3. Due to its manageable complexity, this architecture guarantees a quick forward pass, enabling play at around 1000 frames per second. Also, it can be trained on a CPU. Unfortunately, this training process proved unstable to the extent of taking 10 hours to complete.

To alleviate this, we constructed a convolutional neural network, presented in Figure 4. This design ensured a robust learning process, throughout most of which the running reward increased strictly monotonically. However, backpropagation on this network is computationally expensive and, in consequence, training it is feasible only on a GPU. Additionally, forward passes on the CPU take 5 times longer when compared to the fully connected net. In the end, this network was able to solve Pong after roughly 2.5 hours of training on a CUDA-enabled GPU.

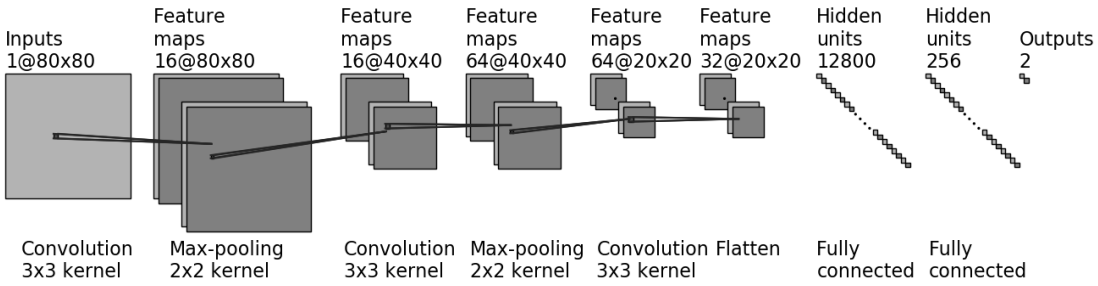

Figure 4: Our convolutional policy network[1]

5 Policy Gradient Methods

Recall that the neural networks in the previous section are called policy networks. This is because they output a probability of moving the player paddle upwards in a given state. Since we decided to constrain our action space to moving the paddle upwards or downwards, it is binary and this output corresponds to a mapping from states and actions to the probability of taking that action in that state. In this sense, our network, parametrized by the set of its weights ω_i, represents a policy:

$$\pi_\omega : \mathcal{S} \times \mathcal{A} \to [0, 1]$$

with

$$\pi_\omega(s, a) = P(a_t = a \mid s_t = s)$$

In consequence, the return we expect from playing Pong based on the output of our network is determined by our network weights. We can now regard the expected return $\mathbb{E}_{\pi_\omega}\big[R_t\big]$ as an objective function of the network weights. Searching this space of parametrized policies directly with gradient ascent by changing the parameters using an estimate of $\nabla_\omega \mathbb{E}_{\pi_\omega}\big[R_t\big]$ is the common denominator of policy gradient methods, which we use to train our networks.

An example of such a gradient estimate is

$$\nabla_\omega \mathbb{E}_{\pi_\omega}\big[R_t\big] \approx \nabla_\omega log \pi_\omega(s_t, a_t)[R_t]$$

which was applied in the *REINFORCE* algorithm [8]. Williams also noted that using a baseline $b(s_t)$ to scale the gradient reduced the variance of the estimate. We put this into practice by modifying our neural networks to produce an additional scalar output, which we train to approximate the expected return following a given state, called the state value function:

$$b(s) \approx V^\pi(s) \coloneqq \mathbb{E}[R_t \mid s_t = s]$$

This way, $[R_t - b(s_t)]$ scales the gradient estimate by an approximation of $A(a_t, s_t)$, the advantage of taking a_t in s_t. This kind of architecture is commonly called actor-critic, albeit we refined this term following a suggestion from [3] by using generalized advantage estimation [4]:

$$A(a_t, s_t) \approx \tilde{A}_t = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta^V_{t+k}$$

[1] This figure is generated by adapting the code from https://github.com/gwding/draw_convnet

with

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

We retain $\gamma = 0.99$ and set $\lambda = 1$.

6 A3C

Training our agent using the aforementioned $REINFORCE$ algorithm is very time-consuming as network parameter updates can only be performed at the end of each Pong episode. In particular, during training phases in which the agent acquired sufficient technique to block returns but is not strong enough to quickly outmaneuver the opponent, an individual rally can take up to tens of seconds and a whole episode often more than two minutes. Instead of adjusting parameter updates after each point, which also takes time, we decided to operate on fixed-length segments of experience instead. However, this means that trajectories containing purely zero rewards are frequently collected. These problems are considered by the $A3C$ algorithm [3]. Here each sequence of rewards is augmented by an estimate of the state value of the last state, given that this is not terminal, before discounting. In addition to this bootstrapping, $A3C$ asynchronously updates global network parameters using gradients computed for agents playing with their own local parameters. After each episode, these agents are reset to the global policy. We omit this feature and instead play multiple independent environments in parallel with the global policy. Every $t_{max} = 21$ steps, trajectories from all environments are accumulated in a single batch and used to calculate losses and update the network weights after computing discounted returns and advantage estimates. This way, we hope to prevent improving play in one area of the state space at the cost of deteriorating performance in other regions, as the original $A3C$ achieves with asynchronism. In fact, there is an on-going debate on whether this asynchronous updating is pivotal to the success of $A3C$. The MinPy documentation also drops this property [2] and articles like [5] describe a similar approach called $A2C$. Nevertheless, we relied mostly on [6] for a great explanation of $A3C$ and do not think that the differences in our implementation warrant a different denomination of the algorithm.

7 Results

Training with A3C used in conjunction with the convolutional neural network architecture described previously enabled solving pong with a running reward greater than 20 in less than 2.5 hours when training on a GPU. The algorithm is robust to changes in its parameters as long as the $ADAM$ optimizer and the convolutional net are used, but quickly destabilizes for the fully connected network when not carefully fine-tuned manually. Albeit trained on different computers, differences between the architectures amenability to training are demonstrated in Figure 5 over a similar amount of episodes. These results are in the same order of magnitude as the implementation of $A3C$ in the MinPy documentation. They solve *PongDeterministic* in roughly an hour using a fully connected network [2] but do not detail which CPU they used, nor how all of their hyperparameters were selected. Lastly, Figure 6 shows that $REINFORCE$ cannot compete with $A3C$.

8 Conclusion and Outlook

Reproducing and modifying widely reported experiments with deep reinforcement learning agents on Gym test problems such as Pong turned out to be a very instructional excercise

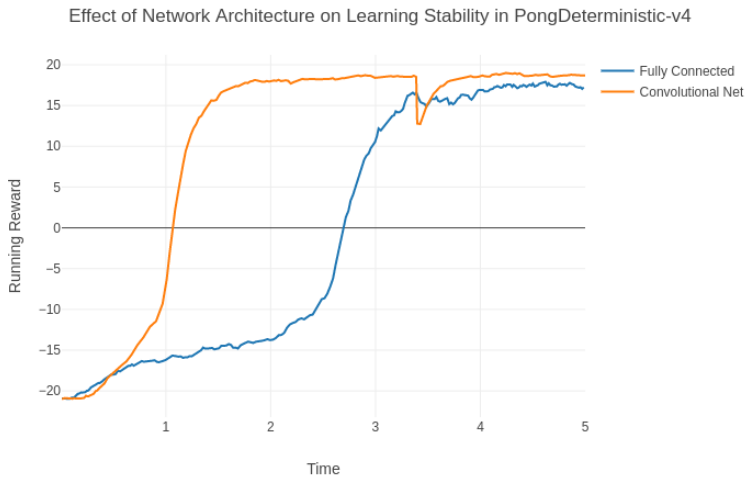

Figure 5: Fully connected versus convolutional net in terms of training stability

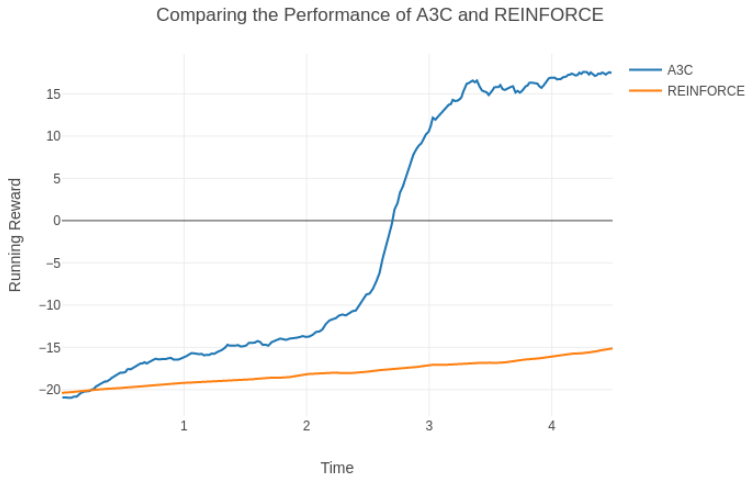

Figure 6: Comparison of the performance of A3C and REINFORCE on the fully connected net

in understanding theoretical background and developing routine in the application of neural networks in reinforcement learning. However, issues related to maladjusted algorithm hyperparameters which are partly interdependent with respect to their effects on learning stability consumed a lot of efforts and proved challenging to resolve methodically. This aspect was disillusioning in terms of our prior expectations concerning artlessness of deep reinforcement learning - even with vast training data on and very clear structure of Pong, our "toy-problem", solving its deterministic version took us weeks. To us, this revealed several issues inherent to deep reinforcement learning, namely the difficulty of reproducing prior work, which goes hand in hand with high reliance on cleverly selected hyperparameters, as well as problematic sample inefficiency, which manifested in having to wait for some hours until we could see whether changes in our implementation could improve its performance. Nevertheless, experiencing that

motivates further engagement in a field in which impressive results and room for improvement seem concerted.

Furthermore, the flexibility of PyTorch encourages its further use. Moving our implementation of $A3C$ to the GPU took only minutes after we had already practiced this for the solver we used for $REINFORCE$. We will gladly make use of this framework again in future work. Also, we found OpenAI Gym to be extremely intuitive and recommend its use without reservations, be it for benchmarking or teaching purposes.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Minpy Documentation. *Reinforcement learning with policy gradient. Link to site.*

[3] Mnih et al. Asynchronous methods for deep reinforcement learning. *ICML*, 2016. arXiv:1602.01783.

[4] Schulman et al. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint*, 2015. arXiv:1506.02438.

[5] Wu et al. *OpenAI Baselines: ACKTR & A2C. Link to site.*

[6] Juliani. *Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C). Link to site.*

[7] Sutton and Barto. *Reinforcement Learning: an Introduction.* MIT Press, Cambridge, Massachusetts, 1998.

[8] Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.