Learning Super Mario using Asynchronous Advantage Actor-Critic

Filip Granqvist, Fredrik Kjellberg

***Abstract*—In this paper, we present our attempt at learning to play Super Mario Bros using a state of the art deep reinforcement learning technique, A3C (Asynchronous Advantage Actor-Critic). The algorithm was implemented and run together with a Super Mario environment. The environment allowed the use of pixels as input as well as tiles. The approach was to train the network with tiles first and then move on to using raw pixels as input. We successfully trained a network with the tiles environment so that Mario was able to complete the first level with ease. Some promising results were achieved using pixels as input, but due to the time constraints of the project, that remains to be completed. Training on more levels is also a future task.**

I. Introduction

Reinforcement learning is widely used in learning tasks where there are neither labels nor training data available. It typically involves an agent that has to interact online with an environment to gather the data, and thereafter evaluate the actions taken depending on the received rewards.

Simple games has been the standard test environments for developing new reinforcement learning algorithms for quite some time and with recent developments such as deep Q-networks (DQN), an agent can now master many of these games [1]. Since the publication of DQN, several algorithms have been developed that outperforms the traditional DQN. This involves algorithms such as Dueling network architectures [2] and A3C (Asynchronous Advantage Actor-critic) [4]. This project focuses on using the A3C algorithm to play the game Super Mario Bros, released by Nintendo in 1985, using only the pixels shown on the screen.

A. Background

We consider the standard reinforcement learning setting where the problem can be defined as a Markov Decision Process. Essentially, there exists an agent that interacts in an environment over a number of discrete time steps. The agent can sense its current state s_t and take an action a_t according to its policy π to transition into state s_{t+1}. When acting in the environment using action a_t the agent receives an immediate reward r_t. Given the discount factor γ and a starting state s_t, the total accumulated return can be described as $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, and the goal of the agent is to maximize the expectation of this formula.

The value function is a fundamental concept in reinforcement learning that uses the above statement to estimate how good it is for the agent to be in a particular state. When following the policy π, the value of a state is defined as $V^{\pi}(s) = \mathbf{E}[R_t | s_t = s]$. A popular area of value-based methods involves estimating the action-value function $q(s,a)$ (Q-learning). Although this project doesn't make use of any of these action-value function learning algorithms, it is good to have in mind that the relation between the value of a state and the action-value is $V(s) = \max_a Q(s,a)$.

Another group of reinforcement learning algorithms are policy-based methods. Typically, these algorithms directly approximate the policy using the input state s and a network parameterized by θ, i.e. $\pi(a|s;\theta)$. A common learning rule is to update θ in the direction $\nabla_\theta log\pi(a_t|s_t;\theta)R_t$ [3].

Asynchronous advantage actor-critic (A3C), is an algorithm developed by Google Deepmind and is currently the state-of-the-art algorithm for most of the popular benchmark games, such as Atari 2600 games [4]. The main idea behind the algorithm is to combine good parts of policy-based methods with good parts of value-based methods. In A3C, the training is also distributed between multiple agents, each interacting with a separate instance of the environment. Each agent trains a separate local network and when enough experience is gathered, the global network is asynchronously updated with the gradients calculated by the agent. After the update, the agent copies the weights from the global network and resumes training. By using several agents, the algorithm is therefore able to asynchronously explore multiple areas of the state-space in parallel which results in more stable learning. A3C maintains a policy $\pi(a_t|s_t;\theta)$ and a value function $V(s_t;\theta)$, both estimated by neural networks. Equation 1 presents the update rule for the policy network, which is similar, but not identical to the update rule presented in the previous paragraph.

$$\nabla_{\theta_p} log\pi(a_t|s_t;\theta_p)A(s_t,a_t;\theta_p,\theta_v) + \beta\nabla_{\theta_p}H(\pi(s_t;\theta_p)) \quad (1)$$

Where:

$A(s_t,a_t;\theta_p,\theta_v)$=	$\sum_{i=0}^{k-1}\gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v)$ - V(s$_t;\theta_v$)
$H(\pi(s_t;\theta_p))$:	is the entropy of policy π
β :	is the strength of the regularizer

Here, the advantage function $A(s_t,a_t;\theta_p,\theta_v)$ is used instead of the plain return R_t, which leads to lower variance estimate of the policy gradient. Also, a fraction β of the entropy of the policy H is added. This prevents premature convergence of the algorithm by requiring high confidence to be able to set the probability of an action close to zero.

A more sophisticated way for defining the advantage and achieving even lower variance of the policy gradient is by using the generalized advantage estimator shown in equation 2 [5].

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+l}^V \quad (2)$$

Where:

δ_{t+l}^V:	is the TD-error at time step $t+l$
λ:	adjusts the bias/variance tradeoff

More specifically, the TD-error is $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$, and is also commonly used as a gradient to update a value function $V(s)$ in temporal difference learning.

B. *Previous work*

There are several attempts to learning an AI master the game of Super Mario worth mentioning. The most famous project being Seth Bling's AI of Super Mario World implemented using a vanilla neural network, genetic algorithms and a tile world as input [6]. He managed to complete the first level, but not the second. Another interesting approach towards learning an AI to play Super Mario Bros was demonstrated by T. Murphy [7]. He derived the reward function from the RAM, assuming that bytes increasing in the RAM is the result of going further to the right and gaining score. To learn the game he used a plain search strategy, running it for thousands of hours, managing to complete the first and second level of Super Mario Bros. There are multiple more AI implementations of Super Mario available online, but none of these seem to have demonstrated a successful run on the third map and beyond on Super Mario Bros.

C. *Purpose*

The purpose of this project is to gain a deeper understanding of deep reinforcement learning as a subject and an understanding of how powerful the current state-of-the-art techniques are, by implementing the A3C algorithm to play Super Mario Bros.

II. Implementation

In this section, the implementation of the A3C algorithm for playing Super Mario is described. Our pre-study revealed that there would be a long training time to receive reasonable results. Therefore, efforts were made to speed up the running time of the environment as much as possible. As a precaution, we first focused on building a working model using the smaller tile world state space provided by the gym environment, and thereafter transitioned into training on raw pixels. The code for this project can be found on Github under user *grananqvist* [8].

A. *Super Mario environment*

The environment used in this project was the Super Mario Bros plugin for OpenAI Gym, developed by Github user *ppaquette* [9]. The Gym environment relies on the Nintendo Entertainment System (NES) emulator, FCEUX [10], for emulating the Super Mario game. Implementing a fast learning procedure was of high priority so the program was set to process only every sixth frame, which decreases the frames the network has to process from 60 fps to 10 fps, which in turn reduces the amount of computation per episode significantly and is still a sufficient amount of frames for the learning algorithm.

B. *Preprocessing of input*

As previously mentioned, two separate experiments was going to be performed: learning to play from a tile world of Super Mario Bros, and learning to play from raw pixels. The tile world can be seen as an extremely pre-processed version of the pixel inputs. The pixel states from the Super Mario environment were the raw RGB pixel values. We wanted to do minimal pre-processing of the pixel states to prevent any injection of human bias. The pre-processing steps that were made was cropping the image from (224, 256, 3) to (176, 256, 3) and normalizing the pixel values. Figure 1 shows how the input image has been cropped.

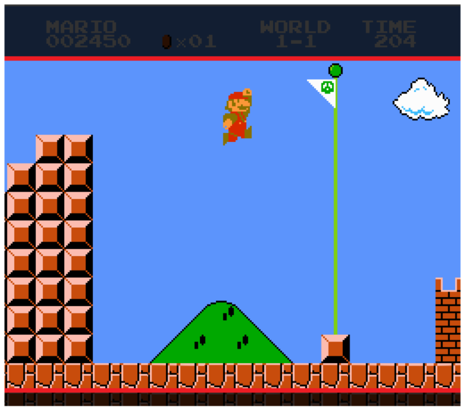

Fig. 1. Image of the screen showing where the input image has been cropped at the top and the bottom. The faded parts are not in the input to the network.

C. *Network architecture*

The network consists of convolutional layers to extract features from the Pixel RGB input. The features goes through a fully connected layer and the output is fed into an LSTM cell. The LSTM is used to capture the temporal aspects of the problem, such as enemy and Mario velocities. The output of the LSTM is flattened and branched into an actor network and a critic network. The critic network is a single output neuron used to estimate $V(s)$. The actor network is a fully connected layer with a Softmax activation function with one output for each action. In Super Mario, there are six possible actions (Left, Up, Right, Down, A ,B), but since some actions can be done at the same time (e.g. Right+A, Left+A+B), we converted this into a one-hot encoding. Hence, the Softmax layer has 14 outputs. Our argument for sharing the earlier layers between the actor and the critic is that they have to learn similar low-level features of the game anyway. The network that we started to test the tile world on before transitioning into training on pixels used an identical architecture but with 2x2 convolutions and stride 1 in the convolutional layers. Table I shows an overview of the network.

TABLE I. OVERVIEW OF THE NEURAL NETWORK ARCHITECTURE.

	Layer type	Properties
1	Conv2d	Filter size: 8x8, Stride: 4, Maps: 16, Input shape: (176,256,3)
2	Conv2d	Filter size: 4x4, Stride: 2, Maps: 32
3	Flatten	Output lenth: 6656
4	Dense	256 units
5	LSTM	256 units
6	Actor Dense	14 units
6	Critic Dense	1 unit

D. Loss and metrics

The loss function that is used consists of three different parts: the actor loss (Equation 3), the critic loss (Equation 4) and the policy entropy (Equation 5). The gradients of the actor loss and policy entropy is identical to Equation 1 explained earlier. The advantage function used in the policy gradient is the generalized advantage estimation (Equation 2). The network parameters are shortened to $\theta_v = \theta_p = \theta$ because of the shared network.

The critic loss is half of the squared TD-error, also presented earlier. The critic is further multiplied by $\frac{1}{2}$ to prevent it from dominating over the actor when updating the shared weights. $V(s;\theta)$ is the output of the critic network, $\pi(a|s;\theta)$ is the output of the actor network, $\hat{A}^{GAE(\gamma,\lambda)}$ is the generalized advantage estimation, R is the return and $R - V(s;\theta)$ together makes the TD-error. The final update rule is presented in Equation 6.

$$L_{actor} = \sum log\pi(a|s;\theta)\hat{A}^{GAE(\gamma,\lambda)} \tag{3}$$

$$L_{critic} = \sum \frac{(R - V(s;\theta))^2}{2} \tag{4}$$

$$H = -\sum \pi(a|s;\theta)log(\pi(a|s;\theta)) \tag{5}$$

$$\nabla_\theta L = \nabla_{\theta_p} L_{actor} + \frac{1}{2}\nabla_{\theta_p} L_{critic} - \beta\nabla_{\theta_p} H \tag{6}$$

In our implementation, we used the discount factor $\gamma = 0.99$, learning rate 1e-4 and $\lambda = 1$ for GAE.

The metrics that we look at are mainly the return and the number of times the level is completed.

E. Rewards

The rewards given to the network are very important in reinforcement learning since it's the only way for the network to get feedback on its policy.

The reward that is given by default by the environment is positive reward for going to the right and equal negative reward for going to the left. A couple of additional rewards were added in order to encourage the network to gather score (e.g. by collecting coins and mushrooms) and penalize dying and not making enough progress. All the rewards can be seen in the following list:

- $+0.03$ for every distance unit to the right.
- -0.03 for every distance unit to the left.
- $+0.01\Delta score$, when Mario gets score (capped at 0.5).
- $+1$ for completing the level.
- -1 for dying or running out of time.
- $-0.01\Delta time$, time decay
- -1 when no progress has been made in the last 30 seconds. (Also kill Mario)

III. Results

The training procedure was performed two times, once using the tile world as input and once using the pixels as input. Because of the limited time available, training was mainly performed on level 1. Figure 2 shows the the return achieved when using tiles as input. In can be seen that the return increases steadily over time.

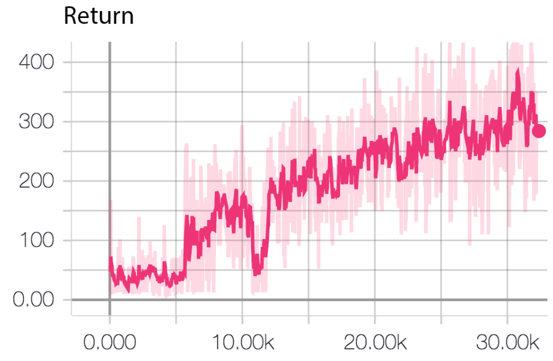

Fig. 2. The return when training with tiles over 30000 episodes.

The performance during training on the larger network using the pixels as state is shown in Figure 3. Due to the limited time available, the network hasn't converged yet as can be seen at the slopes.

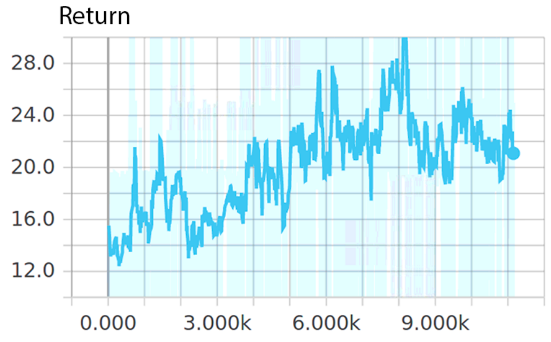

Fig. 3. The return when training with pixels over 10000 episodes.

In Figure 4 we can see how likely it is that the first level will be completed during an episode. This shows that Mario actually manages to complete the level and get's increasingly better at it.

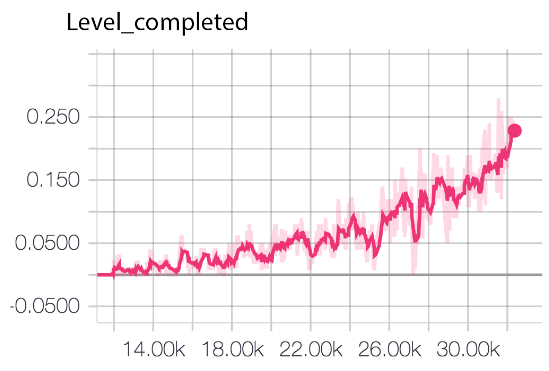

Fig. 4. The likelihood that the first level is completed during an episode when training with tiles.

We have also started to train a copy of the network trained on level 1 to learn how to play level 2, and at the time of this writing, Mario has successfully completed 60% of level 2 at best.

IV. Discussion

The AI performs very well on level 1, much better than what we can achieve. As the performance is still upward sloping at the end of training, one can conclude that more training time would certainly increase the performance further. Also, the training for the network with pixel states was about five times slower, which resulted in an even more undertrained AI. Even though the network with pixel state inputs performs worse at this moment, we expect that when the performance converges for the two networks the pixel network will be significantly better. We support this argument by inspecting the tile world state and can conclude that it is very approximate for Mario and the enemies, and if the tile world was the best pre-processing towards learning the game, the convolutional layers would learn this mapping anyway.

There were some interesting behaviours learned by Mario. Some were due to the game mechanics and some were due to our reward function. Some learned behaviours were jumping with the right force to land on Goombas instead of jumping over them, jumping into bricks to kill Goombas through the roof and even trying to jump as high as possible when reaching the flagpole at the end. The customized reward function turned out to be a bit too biased towards running to the right because Mario discovered quite early that the best strategy for him towards never stopping running to the right while trying to make the optimal jumps to fetch coins and jump on Goombas. This is most likely due to the capped reward on score received, which means that Mario never discover how valuable the hp mushrooms are.

Combining the parallelism of the A3C algorithm with the generalized advantage estimation technique resulted in a very stable learning process for Mario when training on the tile world. Slightly less stable in the start of training for the pixel state inputs, but still manageable.

Regarding Super Mario Bros as a learning environment we can conclude that it is a very good environment if you want to step up a level of complexity relative to the Atari 2600 games. It is also a good environment for learning from sparse rewards, such as mushrooms and the need to diverge from constantly pressing right about 1-5 of the time (not including level 1, its straight forward).

V. Conclusion

We believe that our successful implementation of the A3C algorithm to learn how to play Super Mario Bros have large potential. With some optimization of the architecture and more training time, we believe the AI can reach superhuman performance. Future work obviously includes continuing with training on pixels and also train Mario on more levels than just level 1.

References

[1] V. Mnih, et al. 'Human-level control through deep reinforcement learning', 2015. [Online]. Available: https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf [Accessed: 20- Oct- 2017]

[2] Z. Wang, et al. 'Dueling Network Architectures for Deep Reinforcement Learning', 2016. [Online]. Available: https://arxiv.org/pdf/1511.06581.pdf [Accessed: 20- Oct- 2017]

[3] Yuxi Li. 'Deep reinforcement learning: an overview', 2017. pp. 12 [Online]. Available: https://arxiv.org/pdf/1701.07274.pdf Accessed: 17- Oct- 2017]

[4] V. Mnih, et al. 'Asynchronous Methods for Deep Reinforcement Learning', 2016. [Online]. Available: https://arxiv.org/pdf/1602.01783.pdf. [Accessed: 17- Oct- 2017]

[5] J. Schulman, et al. 'High-dimensional continous control using generalized advantage estimation', 2016. [Online]. Available: https://arxiv.org/pdf/1506.02438.pdf. [Acessed: 21- Oct- 2017]

[6] S. Bling, 'MarI/O - Machine Learning for Video Games', 2015. [Online]. Available: https://www.youtube.com/watch?v=qv6UVOQ0F44. [Accessed: 23- Oct- 2017]

[7] T. Murphy, 'Computer program that learns to play classic NES games', 2013. [Online]. Available: https://youtu.be/xOCurBYI_gY?t=7m42s. [Accessed: 23- Oct- 2017]

[8] F. Granqvist, F. Kjellberg, 'Reinforcement-learning-super-mario-A3C', 2017. [Online]. Available: https://github.com/grananqvist/reinforcement-learning-super-mario-A3C. [Accessed: 17- Oct- 2017]

[9] P. Paquette, 'Super Mario Bros environment', 2017. [Online]. Available: https://github.com/ppaquette/gym-super-mario. [Accessed: 17- Oct- 2017]

[10] FCEUX, 'The all in one NES/Famicom/Dendy Emulator'. [Online]. Available: http://www.fceux.com/web/home.html [Accessed: 17- Oct- 2017]