Homework III - deep reinforcement learning

Neural Networks and Deep Learning
Winter term 2021/2022
Prof. Alberto Testolin

Thursday 7th July, 2022

Nathanael Schmidt-Ott URSNATHANAEL.SCHMIDTOTT@STUDENTI.UNIPD.IT
Home institution: Göttingen University
Guest student at the University of Padova

1. Introduction

This report presents results for the third assignment of the *Neural Networks and Deep Learning* lecture at the University of Padova. In Particular, the goal of this work is to train an agent to solve the Cart Pole game using deep reinforcement learning. The goal of the Cart Pole game is to balance a stick on a cart for as long as possible. The agent has two possible actions to choose from: either she moves the cart to the left or to the right (i.e. the action space $A = \{left,\ right\}$) and the goal is to prevent the pole from falling from the cart. Moreover, the game is also over if the center of the cart reaches the outer borders of the screen. I emulate the Cart Pole game using the Gym environment from OpenAi and provide two approaches to solving the game. First, I use the compact state representation provided by the Gym environment. Second, I train the agent to learn the game directly from the rendered screen (i.e. using the pixel values).

2. Methodology

In each time period the agent observes the current state $s_t \in S$ and selects and appropriate action $a_t \in A$ to transition to to the new state s'. Moreover, the agent receives a reward r_t which depends on the rule of the game. In the Cart Pole game the agent receives a positive reward if the game continues and no or a negative reward otherwise. Moreover, the game is solved if the agent manages to make 500 moves. The action depends on the state s_t and thus we can define $a_t = \pi(s_t)$, where π is the behaviour function or the policy that selects the action, given the current state. In this framework, the environment or the state directly depends on the actions of the agent. Moreover, we assume that the rules of transitioning from one state to the other as well as the rules for reward are stationary (i.e. they do not change). The main goal of the agent is to learn the optimal policy which maximises the expected reward in the long run. The optimal policy can be implicitly learned relying on q-values, which associate a value to a state-action pair:

$$Q(s,a) = \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V^*(s')).$$

$T(s,a,s')$ defines the rules for transitioning from state s to state s' under action a. For instance, this could be a probability. $R(s,a,s')$ is the immediate reward the actor receives after transitioning from state s to s' under policy a. $V^*(s')$ is the expected long term reward from s' onwards assuming that the actor will always choose the optimal policy in the future. Finally, $\gamma \in [0,1]$ is a discount factor which governs the importance of future rewards for the actor. In other words, q-values define the total expected reward if the actor transitions to a new state s' and acts optimally from thereon. Q-learning allows to directly learn these q-values, following

$$Q^{new}(s,a) \leftarrow Q(s,a) + \alpha[r + \max_{a' \in A} Q(s',a') - Q(s,a)],$$

where Q(s,a) is the current q-value, α is the learning rate, r is the immediate reward and $\max_{a' \in A} Q(s',a')$ is the estimate of the optimal future value. The optimal q-value function can be approximated by a deep neural network such that

$Q(s, a, \boldsymbol{\theta}) \approx Q(s, a)$. Thus, the objective function depends on $\boldsymbol{\theta}$:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{s'}[r + \gamma \underset{a' \in A}{max}\ Q(s', a', \boldsymbol{\theta}^-) - Q(s, a, \boldsymbol{\theta})]^2.$$

r is the reward, while the second term is the so-called target network and is updated every C iterations. The third term is the prediction network, optimised at every training iteration. During training, only the parameters of the policy network are learned using gradient descent, while the parameters of the target network are simply copied from the policy network every C episodes. Updating the target network only very C episodes introduces stability to the algorithm and allows to explore the environment in a more stable way uncorrelating the objective function and the Q-network.

Another issue in Q-learning is that the training data is no longer independent and identically distributed as usually in machine learning. In fact, the training data is highly correlated as the observed q-values depend on prior decisions. To uncorrelate the training data, researchers have introduced experience replay, which essentially stores previously learned episodes and then samples from this storage to learn the network parameters of the prediction network. In the replay memory contains N tuples of (s, a, s', r). Finally, there is a tradeoff between exploration and exploitation of the state space. This concerns deciding which action to take. One option is to force the policy network to select the optimal action in every time step. However, especially in early periods it might prove useful to select a suboptimal or random action in oder to explore the state space and to avoid getting stuck in some local minimum. Thus, there is a tradeoff between exploring the state space and exploiting the known best option. Commonly, there are two different approaches to account for this tradeoff: the ϵ-greedy policy simply choses a random action with probability $\epsilon \in [0, 1]$ and the optimal action with probability $(1-\epsilon)$. The Softmax-policy, in contrast, choses an action based on the softmax distribution over all actions. It includes a temperature parameter $\tau \in \mathbb{R}^+$, which governs the tradeoff between exploitation and exploration. $\epsilon = 0$ or $\tau \to 0$ introduce the max-greedy policy, while $\epsilon = 1$ or $\tau \to \infty$ result in a random action. Usually, these parameters are altered throughout the training process to induce a more greedy approach in later training stages. This allows to explore the state space in early episodes and to exploit the knowledge on the state-action space in later episodes. Figure 7 in the appendix show the shape of the decay exemplarily for ϵ. I define it in such a way, that the shape of the decay does not alter, irrespective of how many episodes the model is trained for. Yet, this introduces an additional hyper-parameter δ - the speed of the decay. Higher values of δ introduce a very quick decay, while lower values result in a slower decay. Note, that the target network still uses a max-greedy policy.

2.1 Compact Cart Pole

I first implement the game using the compact state representation provided by the Gym environment. Each state is be described by four variables: cart position, cart velocity, pole angle and pole angular velocity. Thus, the dimension of the state space is 4 and the dimension of the action space is 2. In the first approach I define the policy and the target network as a Multi-Layer Perceptron (MLP) with two hidden layers with 256 and 128 hidden units respectively and a hyperbolic tangent activation function (Tanh) in the hidden layers. As input it takes the four different state variables and it outputs 2 values, one for each action. Based on the outputs of the policy network, the agent chooses the action following some action policy. Here, I explore three different action policies: the max-greedy action policy, where the agent always chooses the best action, as well as the ϵ-greedy and the softmax-policy. I present results for all three approaches. Moreover, I superficially tune some hyper-parameters using the automatic tuning tool Optuna. Specifically, I tune the learning rate lr of the optimiser, γ the discount factor of future rewards, δ the speed of decay, the weight for the position penalty of the cart and the loss function. For the loss function I consider Huber loss, which is a smooth version of the L1 loss and mean squared error (MSE). As optimiser I use stochastic gradient descent (SGD). I set C, the rate at which the target network is updated to 10 after some trial and error and train the agent for a total of 2000 episodes.

Moreover, I set the bad state penalty to -5, use a batch size of 128 and set the maximum size of the replay memory to 10 000.

2.2 Pixel Cart Pole

Rather than using the compact state representation as defined by the Gym-environment, I also train an agent based on the pixel values. At each step, the agent observes an image $x_t \in R^p$, which is a matrix of raw pixel values describing the current state of the game. However, a single image is not able to capture dynamic elements of the game such as the direction and speed of the cart. To provide the agent with respective information, I explore two different approaches: first, I calculate the difference between the current rendered state x_t and the previous rendered state x_{t-1} and it as the input for the agent. Accordingly, $s_t = (x_t - x_{t-1})$. Second, I simply use a sequence of 4 most recent images as input to the agent. In that case, $s_t = x_t, x_{t-1}, x_{t-2}.x_{t-3}$. How many images to include in the sequence is a hyper-parameter, which I set to 4 in this work. Since, the input are no longer scalars, but images, I change the policy and target networks to be Convolutional Neural Networks (CNNs). The CNN consists of two Convolutional layers and two fully connected layers with ReLU activation functions. As in the previous task, the network outputs two values - one for each action. Table 1 in the appendix provides a detailed overview of the CNN used for the policy network.

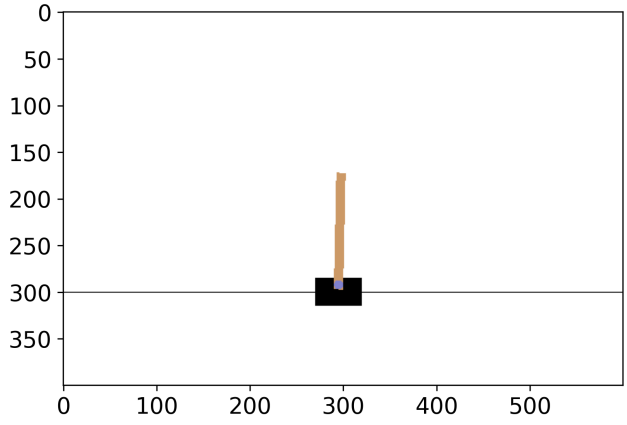


Figure 1: Full RGB image

The Gym environment produces 3 channel (i.e. RGB) 400 x 600 pixel images (see figure 1) to describe the state of the game. Using these images directly would be computationally too expensive and large parts of the image do not convey any important information for the game. For instance, all pixels below the black line in figure 1 or the pixels above the pole are simply white pixels that do not change over the game Accordingly, I crop the image along the vertical axis and along the horizontal axis to capture a subframe containing the cart. Moreover, I transform the images to greyscale images and downscale the images by a factor of θ - a hyper-parameter. I set θ to 0.5 after some trial and error. Figure 2 presents the final input states, that the agent receives. The processed greyscale images measure 80 x 54 pixels.

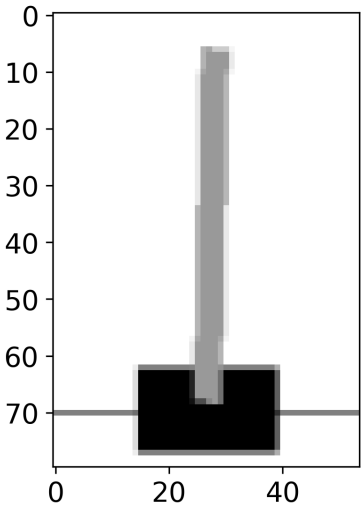


Figure 2: Processed image

After some trial and error, iIuse RMSprop as optimiser with a learning rate $lr = 1e^{-4}$, the Huber Loss function, $C = 20$, bad state penalty of -1, $\gamma = 0.99$, a batch size of 128 and the maximum size of the replay memory to 10 000. Moreover, I train the agent using an ϵ-greedy action policy, starting with $\epsilon = 1$ and decaying it using a decay speed of $\delta = 2$. I use the same hyper-parameters for both approaches (i.e. the first difference and the four frames approach). I train the agent for 1000 episodes.

3. Results

3.1 Basic Cart Pole

Figure 3 presents the 100-episode moving average for the average reward of the agent under for the three different action policies against. In each episode under each action policy, the actor plays until the game is over. Clearly, the softmax and the ϵ-greedy action policy outperform the max-greedy action policy. Under the max-greedy action policy, the agent only very slowly learns how to play the game. Indeed under the max-greedy policy the average reward has not converged even after the agent has played 2000 games. In contrast, under the other two action policies, the average rewards increase relatively quickly and start converging after several hundred episodes, which implies that the agent has learned to solve the game. However, the ϵ-greedy action policy, appears to yield faster convergence to high average rewards compared to the agent under the softmax action policy. Thus, when tuning the hyper-parameters I rely on and ϵ-greedy action policy.

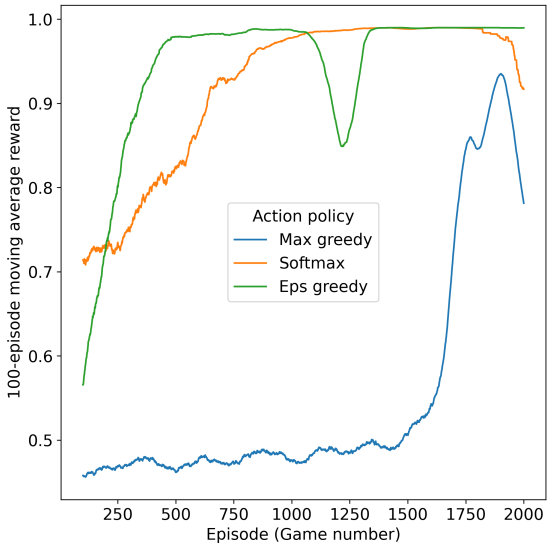


Figure 3: Average reward of different action policies

Indeed, looking at the 100-episode moving average score of the agent under the different action policies, the ϵ-greedy action policy and the softmax action policy outperform the max-greedy action policy by far (see figure 4. Under both policies, the agent arrives faster at higher scores and eventually solves the game. Tuning some hyper-parameters of the model using the automatic tuning tool Optuna and slightly adapting the optimal set of hyper-parameters after some trial and error, the performance of the agent increases further (red line). Though, it takes the agent several epochs, she consistently solves the game consistently solves the game after a few hundred epochs.

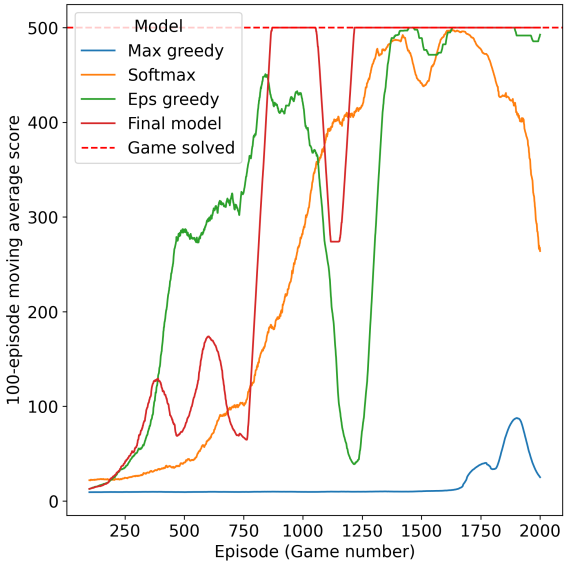


Figure 4: Scores of different models

I tune the hyper-parameters with respect to average reward and select the hyper-parameter combination which achieved the highest average reward over all games. The tuned hyper-parameters are $lr = 0.0098$, $\gamma = 0.9787$, $\delta = 4$, no penalty if the cart is far away from the center (*posweight*) and the Huber Loss function. However, I found the agent to perform better when introducing a linear penalty for the cart position from the center. I alter the model accordingly. I trained the final model for 2000 episodes which took 34 minutes and 4 seconds on a CPU.

3.2 Pixel Cart Pole

Figure 5 presents the 100-episode moving average scores for the two different approaches for the pixel cart pole agent. The blue line represents the score for the first difference approach. In this model the agent receives the difference of the current image and the previous image as input. Training this model for 100 episodes took about 1 hour 2 minutes and 22 seconds on a CPU. Clearly, the agent learns

how to play the game in later episodes. While between the 100th and the 200th episode the agent only survives for about 25 moves on average, the agent survives for more than 125 moves between the 900th and the 1000th episode. Moreover, the curve suggests that the agent has not yet completed training after 1000 and that training the agent for longer would probably increase the average score further. However, the 4 frames approach, i.e. the approach where the agent considers the 4 most recent frames appears to yield better results. In particular, the average score of the agent increases earlier and also on average it achieves a higher score. The 100-episode moving average even reaches a score of more than 175 in the later episodes. Again, it looks like playing more episodes would increase the performance further. However, it took already 2 hours and 20 minutes to train the agent on a CPU and training for longer is beyond the scope of this work. Finally, I set the hyperparameters after some trial and error - tuning these empirically might increase performance further.

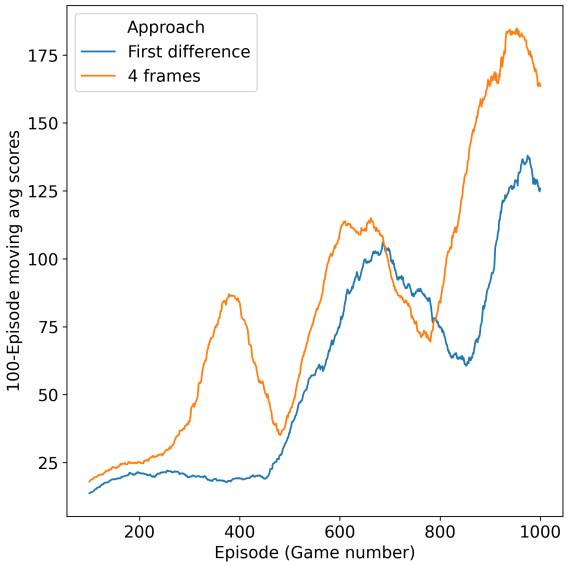
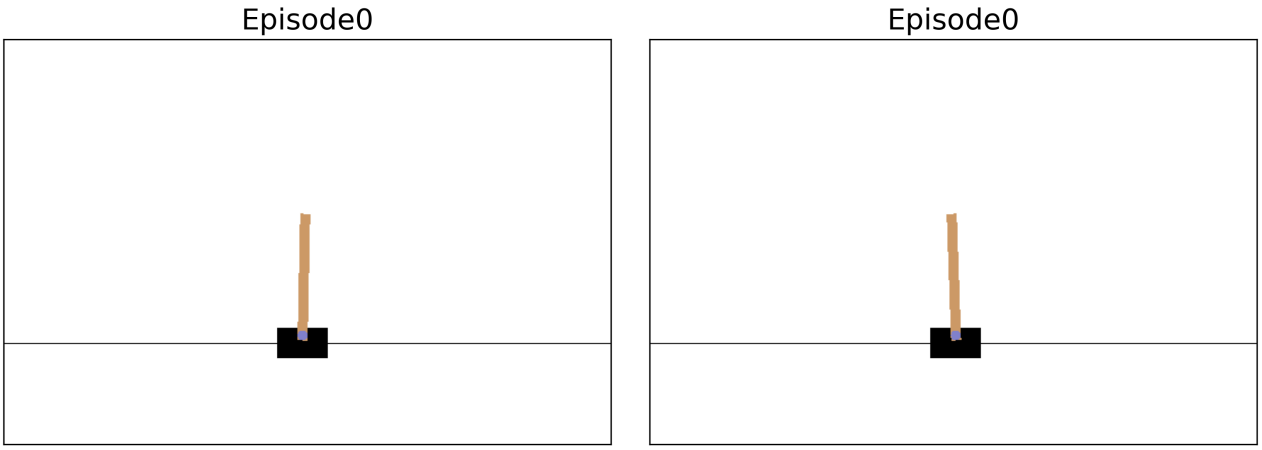


Figure 5: Result of the pixel cart pole

3.3 The final game

Using the learned and best-performing policy approximation functions from both the compact cart pole game and the pixel cart pole game, I let the agent play one episode to compare the different approaches. For the compact cart pole game, I use the final policy network with optimised hyperparameters. For the pixel cart pole game, I use the agent which takes the 4 most recent frames as input, as this approach outperformed the first-difference approach in terms of average score. Figures 6a and 6b present animated graphics (GIFs) for the two games. Note that these GIFs will most likely only work in AdobeAcrobat Reader or any other software that supports animated graphics. The compact cart pole agent played for 500 moves, thus she solved the game, while the pixel cart agent played for only 100 moves. Since, the animated graphics would need to process too many images, I just present the first 50 moves for both agents. The compact cart pole agent (figure 6a) keeps the cart in the middle of the frame and hardly seems to move. This is likely due to the linear penalty that I introduced to penalise the agent if the cart moves from the center of the frame. At first, the pixel cart pole agent seems to learn a similar strategy. However, after a few moves, the cart moves towards the left of the frame. In this particular example, the game eventually ends because the cart reaches the edge of the frame. In contrast to the compact cart pole agent, the pixel cart pole agent does not receive any penalty for moving from the center. Instead, the agent has to learn itself that moving too far towards the edges of the frame will cause the agent to loose the game. It might take a long time for the agent to learn this behaviour, because it will have to explore this first and game over due to touching the edges of the frame is a rare instance compared to game over due to the pole falling off. In fact, the preprocessing of the image data ensures that the cart is almost always in the center of the processed image. The only exception is when the cart is close to the edges of the frame. Thus, as long as the agent stays away from the edges, it will receive no information on the position of the cart relative to the edges of the frame. This probably makes it even harder for the agent to learn to stay away from the edges of the frame. A possible solution to solve this might be to provide larger images (i.e. images that capture a larger part of the frame) to the agent. However, larger images will also increase computation times and require greater memory.



(a) The final compact cart pole agent

(b) The final pixel cart pole agent

Figure 6: Agents playing

4. Conclusion

This reports presents the methodology and the simulation results for solving the cart pole game using deep reinforcement learning. Using the compact state representation provided by the Gym environment enables the agent to solve the game even after a relatively short number of training games. Importantly, training is relatively fast as compared to the approach when using the raw pixels as input. Moreover, when using only raw pixels as input, the agent does not learn to solve the game. However, training for longer and tuning hyper-parameters might improve the performance of the pixel cart pole agent.

Appendix A.

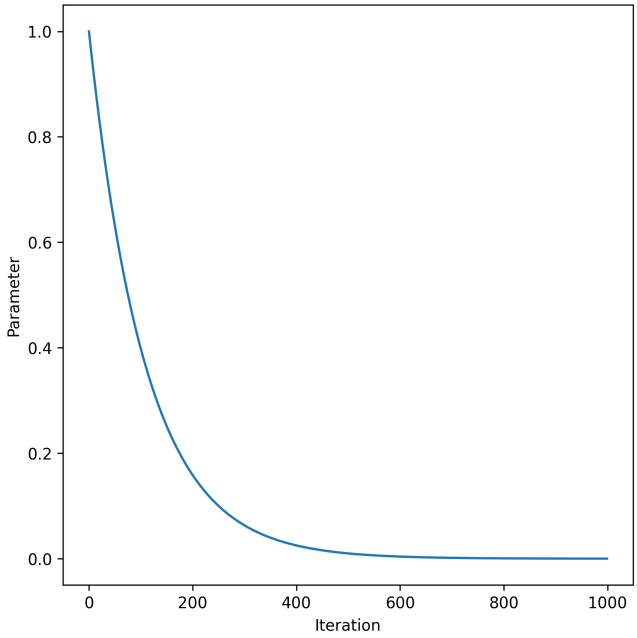


Figure 7: Exemplary exploration profile

Table 1: The ***CNN policy network***

Layer	Hidden units	Input channels	Output channels	Kernel	Stride	Padding	Activation
Conv layer 1		n	16	8 x 8	4	0	ReLU
Conv layer 2		16	32	4 x 4	2	0	ReLU
Linear layer 1	256						ReLU
Linear layer 1	128						