(Extract from) Solving OpenAI's CartPole challenge with Deep Q Learning

Jonathan Collu, Dimitrios Ieronymakis, Riccardo Majellaro

This is an extract from the report for the second assignment of the course Reinforcement Learning (Leiden University). Only Introduction, Background and Methodology are present.

Abstract

There exist a wide range of environments where Reinforcement Learning algorithms can be applied; some of these have a discrete and small enough state space to be approached by tabular methods such as SARSA or Q-learning, while others are more complex with continuous and sometimes high-dimensional state spaces. The problem faced in this report is Cartpole V1 which belongs to the secondly mentioned category of environments, and thus can be approached using Deep Reinforcement Learning methods. We therefore investigated, combined, and compared different model architectures, exploration strategies and approaches to observe which of these combinations provide the best "learning experience" to our agent. In order to properly compare our methods we will mainly focus on the evaluation score achieved, the episodes required to achieve such result and the stability of the training. The experimental results show that DQL is a powerful algorithm, achieving almost optimal results in most of our implementations.

1. Introduction

Many real Reinforcement Learning applications provide an agent to operate in an environment that can have one or both state and action spaces to be continuous. Having an example of these applications makes it easier to understand why it is crucial to increase the effort to study and improve the actual methods to deal with this type of environment (that does not allow having a table to map each state-action pair). Indeed, the most iconic example of Deep Reinforcement Learning (DRL) is self-driving cars, but we can also imagine other challenging applications, such as agents operating in the financial market or playing complex games. For the reasons mentioned above, this report aims to describe and analyze several approaches to solve the gym Cartpole-V1 environment(OpenAI), an environment provided by OpenAI that consists of a cart to be moved along a horizontal axis in a way that allows a pole to stay in a vertical position. In Figure 1, it is possible to observe the environment in one of its possible continuous number of states.

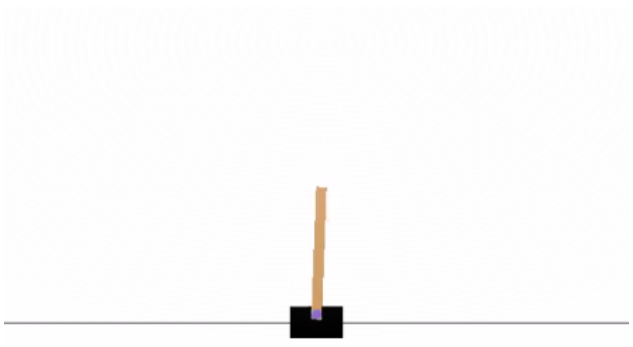

Figure 1. A representation of a possible state of Cartpole-v1. The environment consists of a cart joint with a pole that can be controlled in order to keep the pole vertical. For each time step the extrinsic given reward is +1, and the episode terminates when either the cart is distant at least 2.5 units from the center or the pole is inclined more than 15 degrees from the vertical position. The figure is taken from the Cartpole-v1 documentation(OpenAI).

We investigated, combined, and compared different model architectures (such as MLP and CNN), six different exploration strategies (ϵ-greedy, softmax, UCB, ϵ-annealing, novelty-based, and curiosity-based), several approaches such as training with (or without) a reply buffer, using (or not) a second network to update the target, self-supervised learning, transfer learning, double DQN, and dueling DQN, to observe which of these methods provides the best learning process for an agent in this specific environment. The results obtained by carrying out the experiments suggest that MLPs are architectures that fit particularly well with the addressed problem, obtaining amazing results with the most part of the combinations tested except in the cases where the target network is updated after 100 or 1000 iteration or it uses Upper Confidence Bound as exploration strategy. Furthermore, the results show that CNNs can achieve nice performances with a wide range of settings especially if combined with the usage of Curiosity-based exploration. The results achieved with the CNNs deserve to be highlighted more than the ones obtained with MLPs since it is significantly more difficult to handle states represented as images than states represented by 4 values. Finally, we found that also approaches such as self-supervised learning and transfer learning reach overall good results.

2. Background

Since this report investigates Deep Reinforcement Learning techniques, it would be easier for the reader to understand the following sections after introducing some basic concepts of Reinforcement Learning and deep neural networks. Obviously, these concepts will not be fully addressed here since this is not the main scope of this document. Reinforcement Learning is a branch of artificial intelligence that aims to study and produce methods to make artificial agents be able to learn a good policy to operate (in a reasonable way) in an environment. RL algorithms usually consist of multiple steps (iterations) in which an agent, given its current state, selects and takes one of the possible actions following its actual policy to reach the next state. At this point, using the reward received from the environment, it updates its policy (it basically learns from the feedback it receives). The policy is updated following a formula that can be different for each RL algorithm. Just to provide an example, the well-known Q-Learning algorithm updates its policy as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$$

where s' indicates the state reachable from s taking the action a, and the arbitrary a' is an action which can be taken from the state s'. The reward is indicated by r, while γ is the discount factor which represents the important difference between future and present rewards, and α refers to the learning rate. For what concerns deep neural networks, we used different architectures such as Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), and AutoEncoder (AE). An MLP is a neural network architecture composed of one or more hidden layers (the number of hidden layers is usually small to avoid problems such as overfitting, gradient vanishing/exploding, etc), where each neuron of a layer is connected to all the ones in the previous layer. The nodes in the structure can be "activated" by different non-linear functions. The network is trained using optimization algorithms such as Stochastic Gradient Descent (SGD) coupled with backpropagation to compute the gradient of the loss with respect to the network parameters.
In order to use images to train our Q-network, an MLP is no longer useful, and more sophisticated architectures have been used. One of these is the already mentioned CNN, which is a network composed of a series of convolutional filters used to grasp relevant features, usually followed by fully connected layers ending in the desired output shape. Finally, AEs are structures that, given an input such as an image, encode (encoder) the relevant features into a lower-dimensional latent vector, and then try to reconstruct (decoder) the original input from the latent space. This kind of architecture finds one of its best applications in image denoising.

3. Methodology

A common approach to many reinforcement learning algorithms is to iteratively update a tabular representation of the state-action values $Q(s,a)$ by using the Bellman equation 1 below:

$$Q^*(s,a) = E_{s'\sim\varepsilon}\left[r + \gamma \cdot \max_{a'} Q^*(s',a') | s,a\right], \quad (1)$$

where ε represents the environment, and the remaining notation is equal to the one defined earlier.s Although this approach is bound to converge to the optimal value function, it is unfeasible in practice to represent large and continuous state-action spaces in a tabular form. In order to overcome this complication we can use the $Deep\ Q\ Learning$ (DQL) algorithm. DQL utilizes a deep neural network to approximate the optimal action-value function as:

$$Q(s,a;\theta) \approx Q^*(s,a),$$

where θ indicates the weights of the deep Q-network (DQN). In order to train our DQN we calculate at each iteration i the loss functions $L_i(\theta_i)$ as:

$$L_i(\theta_i) = E_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \quad (2)$$

where

- the target y_i is defined as:

$$y_i = E_{s'\sim\varepsilon}\left[r + \gamma \cdot \max_{a'} Q(s',a';\theta_i | s,a)\right] \quad (3)$$

- $\rho(s,a)$ represents the probability or $behaviour$ distribution over state s and actions a.

- θ_i are the parameters at the current iteration.

In order to optimize the loss function we could differentiate equation 2 with respect to the weights, however this would be too computationally expensive. A more sophisticated approach, is to optimize the loss function by stochastic gradient descent.

The algorithm described above is essentially a $model-free$ algorithm. This means that the reinforcement learning task is solved directly by considering the observed samples ε from the environment, without making assumptions and estimates over it. It is also considered off-*policy* since the target apply the greedy strategy $a = \text{argmax}\, Q(s, a, \theta)$, while the action selection follows a behaviour policy that allows the agent to explore the state space, like the *egreedy-annealing* that we will introduce later.

Learning from single consecutive samples is not effective due to the strong correlations between the samples. In order to improve our DQL algorithm we can implement a technique called *replay memory*. This method consists in storing sets of the agent's experiences, at every timestep i as $e_i = (s, a, r, s')$, in a queue $D = e_1, .., e_N$. These experiences are then sampled randomly in batches (of fixed length), from D and are used instead of the single most recent sample to update the weights of our network. The approach described has the huge advantage of allowing each step e_i to be sampled more than once, over the algorithm iterations. This allows for greater data efficiency and decreased oscillations and instability during training.

Another trick to improve the stability of the model training is to use the so called *target network*. The target network has the same architecture of the *online network*, differently from it, is not trained at every iteration. In fact, its weights are periodically synchronised (every T timesteps) with the online model. We refer to the target model parameters, at iterarion i, as θ_i^-. The target network is used to calculate the $Q(s', a')$ values in the Bellman equation. The predicted Q-values are therefore used in the backpropagation on the main network to update its weights. Essentially, without the target network, the weight updates of the online network change simultaneously both the $Q(s, a)$ and the $Q(s', a')$ state-action values. This makes training very unstable since we are also modifying our target function at every iteration. Instead, by using a target network our model is able to train more efficiently since the target variables $Q(s', a')$ will be fixed for a few iterations, until we update the weights of our target model. Additional details are provided in the original Deep RL paper (Mnih et al., 2013).

3.1. Naive CartPole-v1

In this section we deal with solving the cartpole environment by using the four continuous values returned by the environment itself when we call the *environment.step*() function. Specifically, these four values are the cart position, the cart velocity, the pole angle and the pole angle velocity. This version of the problem is very simple since it directly deals with precise information provided to us by the environment itself. Although as we will see later, almost all our configurations are able to achieve optimal results, these experiments set the basic hyperparameters to tackle the Vision CartPole problem later, where we try to solve the reinforcement learning task based purely on images. What follows is a throughout description of the methodologies and exploration strategies implemented to improve the basic DQL algorithm introduced in the section above.

3.1.1. Double DQN

This method aims to reduce overestimation of the Q values in the target, caused by the max operator. To do so, it uses the online network to select the action based on the current policy and the target network for the evaluation of its value. The target at iteration i is updated by the formula

$$y_i = r + \gamma Q(s', \underset{a}{\text{argmax}}\, Q(s', a; \theta_i), \theta_i^-)$$

where r the current reward, s' the next state, and a an arbitrary action. γ represents the discount factor and θ_i and θ_i^- are respectively the parameters of the online network and the parameter target network, where the latter are updated every T timesteps as $\theta_i^- \leftarrow \theta_i$.

3.1.2. Exploration strategies

Epsilon-greedy: The ϵ-greedy is defined as

$$\pi(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \text{argmax}_{b \in \mathbb{A}} Q(s, b) \\ \epsilon/|\mathbb{A}|, & \text{otherwise} \end{cases}$$

where π represents the action selection strategy given a state s, while $\mathbb{A}$ is the action space. Therefore, a random selection with the goal of exploring is applied with probability ϵ (exploration parameter), while a greedy selection (action with highest Q value) with the goal of exploiting is applied with complementary probability 1-ϵ. ϵ can be tuned to control the trade-off: when it tends to 0 the probability of choosing the best action becomes 100%, while when it tends to 1 the action is always uniformly randomly picked.

Softmax: The Boltzmann policy is instead defined as

$$\pi(a|s) = \frac{\exp^{Q(s,a)/\tau}}{\sum_{b \in \mathbb{A}} \exp^{Q(s,b)/\tau}}$$

where τ is called temperature parameter. This method produces a probability distribution where, even if the probabilities of high values are pushed up and of low values down, there is still a chance of selecting a non-optimal action for exploring. As ϵ, τ is used for the exploration-exploitation trade-off: when τ tends to 0 the policy tends to be greedy, while when it tends to infinity the strategy becomes a uniform random selection. The reason is that all the exponential terms tend to 1 (as e^x for $x \rightarrow 0$ tends to 1), thus each probability becomes $1/|\mathbb{A}|$.

UCB: Instead of using a constant exploration parameter (such as ϵ for egreedy and τ for softmax), Upper Confidence Bound (UCB) is an action selection technique that provides a high exploration when there is not enough knowledge about the environment and increases the level of exploitation as the familiarity with it becomes more and more consistent. The action is selected through the formula

$$a' = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{log(t)}{N_t(a)}} \right]$$

where a' indicates the action to be selected, $Q_t(a) = \frac{r(a)}{N_t(a)}$ (defined as the the sum of the rewards obtained by action a at time step t divided by the number of times a has been selected at time step t) is the estimation of the value of the action a at timestep t, c is a hyperparameter used to control the exploration level (the higher it is, the more exploration we have), and $N_t(a)$ counts how many times action a has been chosen at time step t. Intuitively, $Q_t(a)$ is the factor that ensures exploitation since it prioritizes actions with higher estimated values. The rest of the equation controls exploration based on the level of uncertainty, indeed the smaller $N_t(a)$ is, the higher the exploration term will be (trivially uncertainty increases the need for exploration), while if an action has been taken multiple time the confidence level raises (based on the number of times the action has been selected) and the exploration term becomes smaller.

Novelty based: This strategy encourages the discovery of novel states to ensure a broader exploration and thus a more consistent knowledge of the state space. In order to describe this method, it is crucial to introduce the concepts of extrinsic and intrinsic rewards. The former type of reward simply consists of the rewards that the agent can normally obtain from the environment, whereas the latter consists of an intrinsic bonus to be added to the extrinsic reward in order to guarantee a certain behavior (in this case the exploration of novel states). The current policy is updated using

$$r = r_e + r_i$$

where r_e is the extrinsic reward and r_i is the intrinsic reward both referred at an arbitrary time step. Since the problem addressed has a continuous state space, it was necessary to discretize it to properly implement this strategy since it is based on the rough count of the times state s was visited. To do this is necessary to have a function

$$\phi : S \to \mathbb{Z}$$

that maps the continuous states in hash codes. As proposed by (Tang et al., 2017), we used the SimHash approach which produces a discrete hash code from a continuous space through the formula

$$\phi(s) = sgn(Ag(s)) \in \{-1, 1\}^k$$

where Ag is a matrix where each entry is given by a normal distribution ($\mu = 0$, $\sigma = 1$) and its dimension is given by the product between the state dimension and a scalar hyperparameter k that defines the length of the hash codes. Once the states are mapped into discrete hash codes, they can be stored in a dictionary where the keys are the hash codes and the values are the occurrences of the states relative to these keys during the past time steps. At this point is possible to compute the intrinsic reward as

$$r_i = \frac{\beta}{n(\phi(s))}$$

where β is a hyperparameter representing the bonus coefficient (the higher it is, the most our strategy encourages novel states) and $n(\phi(s))$ is the count of the times the state s has been visited. Trivially, high values of $n(\phi(s))$ produce smaller bonuses to avoid visiting over and over the same state, whereas small values produce higher bonuses to explore less visited states.

3.2. Vision CartPole-v1

With Vision CartPole we refer to approaching the CartPole problem exclusively from an image perspective. In fact, instead of considering the default 4 values provided by the environment as the state, we represent it using only the pixel values from the game frames. This problem is much harder for different reasons. First, there is no explicit information provided to the agent, but a raw sparsely informative tensor of pixel values. Second, the data is high-dimensional; in fact, each frame has a size of 600×400×3, where the 3 refers to the RGB channels. To mitigate this problem, the images are converted to grayscale, as the color is not informative in this case, cropped (200×200) around the cart and downscaled (100×100). This preprocessing is hence helpful to increase the information density in our input. An additional problem occurs when considering a single frame: as the temporal information is missing, it is not possible to derive the velocities of the cart and the pole. We are able to overcome this issue by stacking 4 consecutive frames along the depth dimension. Therefore, the input to our DQN is a tensor of shape 100×100×4, ∼1/18 of the original single frame RGB size.

3.2.1. Dueling DQN

This method is proposed by (Wang et al., 2016) and aims to avoid computing the value of actions when it is not strictly necessary and predict the state value always since it is crucial in every step. For the above reason, the Q-network is implemented as a convolutional neural network followed by two different streams of fully connected layers in order to estimate the state value and the expected value for each action independently. The DQN is defined by the formula

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathbb{A}|}\sum_{a'} A(s,a';\theta,\alpha) \right)$$

where s is the current state and a is the last taken action. θ are the parameters of the convolutional layers, α are the parameters of the fully connected layers used to predict the benefit of taking a certain action, β are the weights of the fully connected layers used. $Q(s,a;\theta,\alpha,\beta)$, $V(s;\theta,\beta)$, $A(s,a;\theta,\alpha)$ are respectively the state-action value function, the state value estimation, and the action value estimation vector.

3.2.2. Self-supervised learning

At the beginning of our experimentation with the Vision CartPole, we obtained poor results which seemed to improve at a slow pace as the epochs increased. We thus concluded that, with the current configuration, it would have been infeasible to obtain near-perfect scores within the time at our disposal. An interesting possible solution to the problem is the use of self-supervised learning with an autoencoder architecture to pretrain the model in a meaningful way, and to finally fine-tune it on the original task. The autoencoder is composed of two parts, the encoder and the decoder. The former takes as input 4 grayscale frames stacked together in 4 channels, and through 3 convolutional layers produces a lower-dimensional latent representation in the shape of 64 stacked feature maps of size 9×9 (overall, is ∼1/8 of the input size). The latter decodes the latent representation using a series of 3 transposed convolution layers with final sigmoid activation functions, having the goal of reconstructing the input tensor. The output has therefore the same shape as the input. We collect a total of 500 random (referred to the action selection procedure) episodes, and for each of them we perform a training step using the binary cross-entropy loss function. After this self-supervised pretraining phase, the decoder is substituted with a fully-connected classification head in order to fine-tune the model on the original problem. The intuition behind this whole process is that the convolutional filters of the encoder could be more easily trained over the simpler task of reproducing the input frames, while still being adequate for the reinforcement learning task. Ideally, the DQL algorithm would then be more stable and require less epochs for reaching meaningful results.

3.2.3. Transfer Learning

Considering the same premise made in the self-supervised learning paragraph, we came up with a second interesting and similar solution to the same problem. The idea is to select a simpler but related task to easily pretrain a CNN, and then fine-tune the model on the original harder task using a different classification head. The information acquired on the first task (hence the learned convolutional filters) would ideally be meaningful for the second one, boosting therefore training time and stability. The (probably) simpler and more closely related problem directly obtainable from the Vision CartPole environment is the following: given a stack of game frames, predict whether the pole is falling on the left (label 0) or the right (label 1) side of the cart. Similarly to self-supervised learning, we collect random episodes and pretrain over them. In this case, however, the target is not the input tensor, but a simple 0 or 1. It is possible to extract it from the last input frame by summing the pixel values on both sides; the side with the lowest resulting sum is the correct one, because the pole is represented by low value pixels (black), which substitute the high ones of the background (white). As in the autoencoder, the loss function is the binary cross-entropy. Once the model is able to tackle this problem, the classification head is replaced by the one used to predict the Q values. The convolutional layers can be either frozen or modified during the fine-tuning phase.

3.2.4. Curiosity exploration

This exploration strategy is performed by using intrinsic rewards to update the policy in such a way that the agent is encouraged to explore less familiar states by increasing its curiosity. Over the years, several approaches to implementing this strategy have been proposed, and for our scope, we chose the "Intrinsic Curiosity Module (ICM)" presented by (Pathak et al., 2017). This method is implemented through a forward model

$$f_{\psi_F}(\phi(s), a) = \hat{\phi}(s')$$

that given in input the feature space of the current state and the action taken by the agent at the actual time step, predicts the feature space of the next state. To be sure that the feature space only contains information related to the action taken by the agent, ignoring the other changes in the environment, ICM makes use of an inverse model

$$g_{\psi_I}(\phi(s), \phi(s')) = \hat{a}$$

that takes as input the feature spaces of the current and the next state to predict the action taken by the agent. The inverse module is trained in a self-supervised way using the action taken by the agent as a target (the loss function used to train the inverse model is the Cross-Entropy), whereas the loss function used to train the forward model is defined as

$$||\hat{\phi}(s') - \phi(s')||_2^2$$

The above formula measures how the agent is able to predict the consequences of its actions and thus the higher it is the lower is the agent's knowledge about the next state. For this reason, it is used as an intrinsic reward to improve the agent's curiosity.

References

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

OpenAI. Cartpole-v1. https://gym.openai.com/envs/CartPole-v1/.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787. PMLR, 2017.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Xi Chen, O., Duan, Y., Schulman, J., DeTurck, F., and Abbeel, P. # exploration: A study of count-based exploration for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pp. 1995–2003. PMLR, 2016.