Deep Reinforcement Learning Beginners Tutorial Documentation

Julian Bernhart, Robin Guth

June 21, 2019

Contents

1 Introduction

Deep Reinforcement Learning (also called RL) is a huge step towards the creation of an universal artificial intelligence. In 2013 a company, owned by Google, called "Deep Mind", was able to create an astonishing implementation of RL, which was capable to play retro games of the console "Atari 2600". In many cases the AI (Artificial Intelligence) was not only able to play the games successfully, but also exceeded human performances significantly. After these impressive results, it is definitly worth to take a closer look at Reinforcement Learning.

legitimation: Playing games is fun, but Reinforcement Learning has more to offer, as discussed in [3] . Apart from playing video games, there are use cases in many fields like robotics, traffic control or even personalized advertisement. While supervised and unsupervised learning are already used widely in production, reinforcement learning is still in development and further research is needed. As a fairly new topic, beginners often struggle to find a good starting point into the world of AI and specifically RL. Many tutorials are written for more advanced users, who already have a deeper unterstanding of machine learning. The "Deep Reinforcement Learning Beginners Tutorial" will provide an easy-to-follow, hands-on beginners guide to RL. After the completion, we will be able to write our own algorithm to play some basic games for us.

Before we head into the world of Reinforcement Learning, we will have to talk about software agents.

goals: Why game?

notebooks installation guide /notebooks -¿ Startschwierigkeiten -¿ guide

Requirements

2 Deep Reinforcement Learning

In this section we will discuss the content of two developed notebooks, mainly the theory behind Deep Reinforcement Learning (DRL). We will get to know the basic concept of RL, see on an exemplary neural network model the deep learning aspect and work out the necessary formulas in order to translate the theory into code. Further we will introduce a framework, which helps building this project by providing a game to play, more on that later. All the theory, explanations and code can be viewed in completion in the enclosed notebooks:

1. Deep Reinforcement Learning Beginners Tutorial (1) - Theory
2. Deep Reinforcement Learning Beginners Tutorial (2) - Practice

motivation

2.1 The Software Agent

Befor we start with the topic of RL we have to adress some basics in order to clarify the following explaination. We will use the term of *software agent*, which, as part of artificial intelligence, is a program, that acts self sufficient to solve a task. There are three important aspects an agent should fulfill. First are autonomous actions. An agent needs to make decisions without external help. Further an agent needs to execute multiple actions to complete its task, so it should be proactive. Last but not least, an agent has to be reactive, which means it reacts on changes in the environment it is in. An optional ability for an agent like this is independent improvement. For this the agent builds up knowledge after repetitively doing its task and improves itself this way.
Basically, the part of the agent, which controls its actions, can be filled with different algorithms. In our case, this will be an implementation of RL, which fulfills all requirements of an agent.

2.2 Reinforcement Learning

Reinforcement Learning is considered one of the three machine learning paradigms, alongside supervised learning and unsupervised learning. The main goal of RL is to create an agent, which can navigate by actions in an environment in a way to maximize a representation of a cumulative reward. The evironment is a space that contains the agent. We will talk about the environment later on. In order to miximize the reward, the agent has to learn by trial and error which actions most likely lead to a reward and which lead to a penalty. After some training, the agent will use its knowledge to avoid previous mistakes. It still has to explore the environment, because otherwise we can not be sure if we actually find the global maximum or just a local one and if there might be a better chain of actions. If we think about our real world, this method of learning is pretty close the human learning. If we get hurt for example, we are more likely to try to avoid the situation. Still curiosity or necessity leads us to exploring and thus maybe getting us into danger, but in the best case, will lead us to some kind of positive reward.

2.2.1 The concept of RL

Our agent is contained by an environment. A momentary snapshot of the environment is called state. A state contains all information at a certain point in time for example the position of the agent or enemies. There is a permanent exchange of information between the agent and its environment. The agent receives the actual state and has to choose an action based on its logic. Everything the agent can use to alter its environment is called an action. It changes the environment based on a certain set of rules. After executing an action the agents receives the new state and a reward, which helps to decide whether its decision leads to a positive result. Keep in mind, that a reward can also be negative. Basically, an environment is just a set of states. We can move between states by executing actions. A good example for an environment is our own world. We are agents, moving in the world. All the time we have to look at our surroundings and choose an action like moving accross the street or waiting until it is free. The laws of physics restrain us, take gravity for example. We receive rewards like geting hurt oder feeling satisfied, thats how we evalute our actions. Our environment is constantly changing, so we have to reevalute our decisions and choose an action again. The following picture shows the whole procedure:
Our goal is to learn the best action for each state, so that our total reward reaches its global maximum.
All components such as reward, amount and type of actions or states depend on the type of environment and can vary widely. As the real world example is pretty complex, for the rest of the notebook we will use a simple game environment for explanation. The informations available for an agent are the visual output of the game (actual state), the inputoptions and the reward for a previously taken action.
The following diagram shows an exemplary setup for a RL project with a game as an environment. For the navigation through this environment the agent needs the actual state (game screen) as an input, in order to respond with an appropriate action. Every action taken leads to a value added to the score of the agent and a change in the environment resulting in the next state of the game.
In order to improve its knowledge, the agent has to repeat the task multiple times. For this we define a set amount of epochs with a set amount of time for the agent to play.
This is where the RL part comes into play. In order to maxmize the reward, the agent has to learn which actions result in a positiv reward and should be prefered and which ones should be avoided because of a penalty. In the beginning, the agent has no knowledge about the environment and thus cannot make an educated desicion. It will need to explore by taking random actions and observing the reward. These informations will be added to the knowledge

of the agent. Over time the agent needs to decrease the amount of explorational actions and increase the amount of exploitational action in order to progress. This means that the agent will start to decide on an action based on its gained knowledge, while decreasing the amount of random actions. A good decreasing ratio is very important for RL, as changing to fast will lead to bad results, while changing it to slow will lead to overfitting.

2.2.2 Deep Learning Aspect

The agent has to recognize and differentiate elements in the environment, to be able to act accordingly and choose a proper action. In the best case, we already have the input reduced into some important numbers, which describe the actual state. In real applications this may not be possible or feasible with our knowledge. Most of the time we will have a picture available for example the image of a camera or a game screen, but the input we receive can basically be any sensory data.

Furthermore the agent must associate the occurence of individual elements on screen with its own actions and the subsequent reward or penalty, which may occur only after several steps in the future. Depending on the situation it may be also necessary to estimate what the next state of the game could be. There are two possible ways to use the Deep Learning aspect.

1. Image processing with a Convolutional Neural Network (CNN), in order to process the image information and send this to the agent.
2. A Deep Q-Learning Network (DQN), in order to model the Q-function, which is discussed below.

Following is an exemplary presentation of a neural network, which is designed according to the tasks just mentioned.

There are two ways to learn in deep reinforcement learning. Eiher we can use Value Learning or Policy Learning. With Value Learning, we assign values to each state-action pair, which correspond to the probabilities of the pair beeing the best option. In Policy Learning, we will learn a strategy instead, which gives as the best estimated action for a given state s. We will focus at Value Learning.

2.2.3 Q-Learning

The task of the agent is the maximization of a cumulative future reward. An example of this can be the score in a game. In an environment, there is no guarantee for an immediate reward after executing an action. In many cases, only specific chains of actions lead to a positive reward, so the agent needs to learn multiple moves in succession in order to fulfill the given task. For example, if the agent needs to collect an object in order to get a higher score, it may be necessary to take multiple moves to reach it. The moves in between may not vary the reward significantly or even reduce it, however they are needed to complete the task in order to be successfull.

A simple way to implement RL is Q-Learning. We assign a numerical value to each action we can execute per state. This value is called Q-value. It represents an estimate of which action will result in the highest reward for the actual state. This is the knowledge of our agent. Each step we have to choose an action for the actual state. This is either random or knowledge based. We use random actions to explore our environment. If we use the knowledge of the agent, the next action is chosen by exploitation, so the agent uses the action with the highest Q-value for the given state.

The Q-Values are not known at the beginning and need to be learned by training. For this purpose, the agent needs to explore its environment. In the best case, it will visit all states through every possible action, but most of the time, this is neither needed nor possible, because of the amount of states or actions. We only desire the path with the best overall reward. We have to keep in mind, that there is no guarantee, that a path we found is the global maximum. It is possible, that the agent is stuck at a local maximum and needs to sacrifice some reward to be able to reach a state with a bigger payout. The Q-values are initialized by 0. After every step the agent takes, we need to update the last used Q-value according to the reward the agent receives. If the agent loses, dies or does something else we want to avoid, the resulting reward is negative. Either positive or negative, the Q-value needs to reflect this with its value. We need to update the Q-value regardless if the agent is exploring or exploiting. The exploration rate ϵ is slowly descending while training, so the agent can use its gained knowledge to improve the reward.

In order to adjust the Q-values we need the total reward:

$R_t = \sum_{i=t}^{\infty} r_i = r_t + r_{t+1} + ... + r_{t+n} + ...$

The *total reward* R_t is the sum of the actual reward and all future rewards from actual state t to state n = ∞.

We also add a discount factor γ to this formula. That factor has a range of value between/from 0 to 1 and is used to decrease rewards that are further into the future. This makes rewards in near future more dicisive for the agent.

$R_t = \sum_{i=t}^{\infty} \gamma_i r_i = \gamma_t r_t + \gamma_{t+1} r_{t+1} + ... + \gamma_{t+n} r_{t+n} + ...$

The total reward R_t is the discounted sum of future rewards.

After dicussing the Q-values our next step is the associated function. The Q-function is the name giving aspect of a Deep Q-Learning Network (DQN), while the "Q" stands for "Quality". The higher the reward, the higher the estimated Q-value and associated quality. This function represents the expected total future reward an agent could reach by executing a certain action a_t in a given state s_t and is used for our deep neural network. The DQN takes over the task of estimating this Q-value. With no prior information at the beginning, the agent needs to extends its knowledge by trial and error and thus learning, how to determine a Q-value out of the actual state and an action.

<Q-Value depending on s and a >=<Expected total future reward>

$Q(s, a) = E[R_t]$

If we have a good approximation of the Q-values for each state-action-pair after the exploration, the agent needs to decide on which action is the best for a given state. This means the agents needs to choose the action which maximizes the estimated future reward by the associated Q-value, with the help of a policy

$\pi^*(s)$. This is the part of exploiting the gained knowledge.

<Optimal policy depending on s>=<Action resulting in the highest future reward according to current knowledge>

$\pi^*(s) = \underset{a}{\text{argmax}} Q(s, a)$

Finally, we will take a look at the formula, which we will actually use to calculate Q-values. At the moment we use the current state of the environment, the actions to take and the reward, resulting from the decision of the agent as input. We also use the following state of the environment. We introduce the *Bellman equation* now:

<Q-value depending on s_t and a_t >=< Reward observed after the last action (immediate reward)> + < Discount factor > $*$ < Maximum Q-value for the next state of the game (future reward)>

$Q(s_t, a_t) = r_t + \gamma * \max_{a'} Q(s', a')$

For now the Bellman equation provides the Q-value for a given action in a given state. In order to train our DQN regarding the Q-function, it is necessary to compare the Q-values predicted with the actual ones and minimize the error between the two. If we use the Bellman equation to calculate this difference, by simply subtracting these values, we get the so called *temporal difference error.*

<Temporal difference error>=< Target Q-value> $-$ < Predicted Q-value>

$\delta = (r_t + \gamma * \max_{a'} Q(s', a')) - Q(s_t, a_t)$

To update an old Q-value, we use the temporal difference error and add it to the Q-value. A new discount factor, called learning rate α, determines the weight of the temporal difference error.

<new Q-value depending on s_t and a_t >=<old Q-value depending on s_t and a_t > + <Temporal difference error>

$Q(s_t, a_t)_{new} = Q(s_t, a_t)_{old} + \alpha * \delta$

Now we use *gradient descent*, an optimization algorithm, to optimize the Q-value approximation. Therefor we can train the DQN using *mean square error* as the loss function, which results in the the following formula.

$L = E[\|\delta\|^2]$

2.2.4 Model Q-function

The following illustration shows two options by which we can model our Q-function with a deep neural network. The model on the left side estimates one Q-value at the time. This is what we are using at the moment. The model on the right side is the prefered one, because it increases the efficiency. It gets the state of the environment and calculates multiple Q-values, so we just have to choose the highest value to determine the action for the agent instead of estimating the Q-values for every action individually. This will save processing power, as the amount of possible actions can be very large.

2.3 The OpenAi Gym Framework

2.4 The Exercises

3 Methods and Materials

keras etc jupyter lab open ai gym: As we discussed before, Reinforcement Learning can be used to solve a range of different problems. Developing Machine Learning algorithms is often not easy to understand nor comprehensible especially for beginners. Furthermore, it is important to be able to compare the performance of different iterations of our algorithm, to be able to improve it.

So bascially we need an environment, that we can use to test and train our RL agent, which fulfills the following requirements:

repeatable test/training epochs finite set of inputs finite set of actions easy state representation easy to control agent deliver a score for a given state

In practice, not all of these points will be fulfilled, but as this is a beginners guide, we will start with a simple environment. Luckily, many video games can be used as quite good environments for machine learning purposes. Many implementations of RL are tested with games as Benchmark and there are some good reasons for this. Developing a whole test environment would be labour intensive and would require dedicated work towards a useable simulator. Using an existing game is also easier to compare to human performance and therefore the evaluation of different algorithms is easier. Another important point is the size of possible inputs and actions. The AI replaces the human player. Depending on the game, the input for our agent is an image, like a human player would see it. The set of actions is a combination of different buttons, which can be pressed on a controller. Finally, games are fun and most people can relate to them. It is also easier to understand what we want to accomplish, because we can transfer aspects from our human play style to the behaviour of an AI.

4 Results

critic/ expensions: using a framework policy learning / value learning -¿ both variants as cartpole agent policy gradient

notebooks: theory, exercises installation guide

outlook