How To Play Tetris With Deep-Q-Learning

Overview

Our initial goal in this project was to get acquainted with the idea of Reinforcement Learning, ideally via coding it with a simple example. We also wanted to learn **TensorFlow**, which is why we programmed the Learning part using Google's library. This is also an indirect excuse for our inelegant implementation. Inspired as many others by DeepMind and their success with letting Neural Networks play Atari-Games, we decided to choose a similar example. Since we both weren't familiar with importing these games directly, emulators and stuff, we chose a game that is relatively easy to implement, Tetris. Writing the game ourselves allowed us to directly implement all the functions, which made the training process easier (but debugging harder...). We followed the approach of **DeepMind**, namely using **Deep-Q-Learning** and directly feeding the raw pixels of our game into the Convolutional Neural Network. I will describe the theory and the architecture used in the next paragraph. As it turns out (so far at least), Tetris was a bad choice (It would have been worth it to check for existing papers before programming the stuff for weeks..). The game structure (obviously) asks for relatively long-term decisions, since you cannot really associate a reward with a decision immediately. Blinded by all the Machine Learning Magic, we thought that it will somehow work, I mean we're using a Convolutional Neural Net, nothing can go wrong, right? After weeks of failure, we started to doubt our implementation. It was very unclear if really Tetris is the problem or if we as coders are the problem. We solved this by implementing yet another game, which we knew was sucessfully tackled by others. We chose FlappyBird and programmed it **very** crudly.

The Theory of Reinforcement Learning

Definitions

Let's first get to dry theory. How the hack is it possible to let an agent learn how to play games??
We first need to introduce some definitions. I will always give an example for these definitions using our implementation of Tetris.

- **Agent**: Our AI that will learn to play the game.
- **Environment**: The system our agent interacts with, sometimes it can be stochastic a.k.a. include a portion of randomness.
- **State**: An instance S_t of the environment that our agent will be able to observe at time t.
- **Action**: An action A_t that our agent can take in order to change the current state.
- **Reward**: The points r_t that our agent gets for changing the current state by performing a certain action.
- **Total Future Reward**: The future points we get at time t:

$$R_t = r_t + r_{t+1} + \cdots + r_n$$

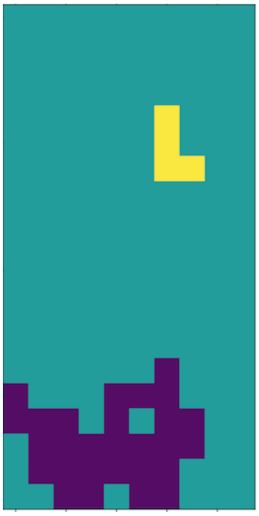
- **Discounted Future Reward**: The future points we get at time t annealing them with a factor of γ:

$$R_t = r_t + \gamma * r_{t+1} + \cdots + \gamma_n * r_n$$

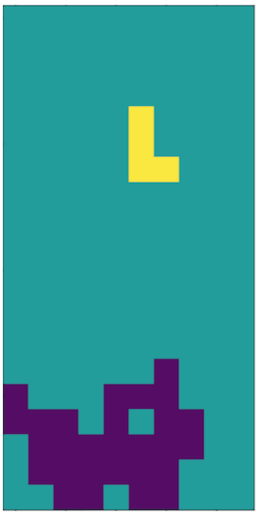
- **Episode**: One complete walk through the environment until the agent reaches a final state.

Don't worry about these rather technical definitions, they are all actually very natural to our problem.
In the case of Tetris, the agent will be our Convolutional Neural Network and
the environment is the game itself with which the agent can interact. It is partly stochastic as we never know what new block will appear at the top when we place the block before on the bottom.

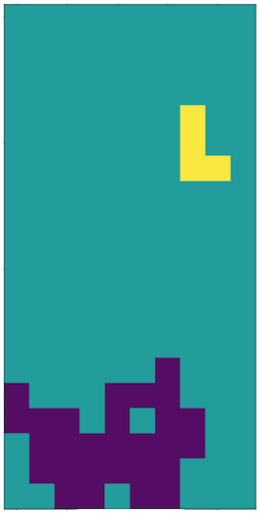
The states are given by snapshots of the game, a.k.a. the image one sees when playing the game and
the possible actions that our agent can perform are **Wait**, **Left**, **Right** and **Rotate**.
The rewards are the points you get when playing Tetris, e.g. for completing a row or when the blocks reach the top.
The total future reward is all the rewards we will collect starting from time t until our agent loses.
The discounted version tries to include the uncertainty in future rewards due to the randomness of our game, letting them count less by multiplying them by $\gamma < 1$.
The episode is simply one round of Tetris until our agent loses. (GameOver)



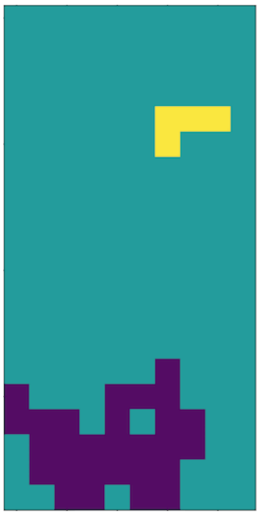
Action "Wait"



Action "Left"



Action "Right"



Action "Rotate"

A complete summary of the game is given by the following sequence:

$$\{S_0, A_0, r_0, \ldots, S_n, A_n, r_n\}$$

It's a collection of all states, actions and rewards that our agent got, ordered with respect to time.

What will be **very important** later is the following observation:

$$R_t = r_t + R_{t+1}$$

In words we can write the total future reward at time t as the sum of the reward we get for

going from S_t to S_{t+1} plus the total future reward at time $t+1$.
Makes perfect sense so far.

Q-Learning

Having these definitions now out of the way, let's get our hands dirty with the actual theory behind Reinforcement Learning.

The goal for our agent is to be able to decide what action A_t it should use at time t, given the current state S_t.
At first glance, this sounds like a classification problem:
Given the state S_t, decide what action A_t is best.
The first surprise is that we will treat it as a Regression Problem!
Imagine the following:
What if we could predict the **future maximal reward** we get if we choose action A_t, given that we are in state S_t?
Namely if we had a function

$$Q(S_t, A_t) = \max R_{t+1}$$

that maps state and action pairs (S_t, A_t) to future maximal rewards, we could always choose the action that maximizes this function and the problem would be solved.
This function is defined on the space of all possible states and actions, which is immensive since the number of possible states is given
by 3_{20*10}. Note that we model our Tetris game as an 20x10 array containing either 1, 0, -1.
This makes it impossible to determine Q exactly. That's where our beloved Neural Nets come into play!

We use a Convolutional Neural Network as a function approximator to our so called **Q-Function**, imitating it better and better with every game our agent plays. But let's have a closer look at how this network will actually learn.

Usually a Neural Net (or any Machine Learning algorithm stemming from the **Supervised Learning Region**) requires targets that it needs to approximate. Think of classification problems such as recognizing dogs from pictures or determining the age of a patient based on his brain scan etc, all these problems require training data, where we know what our true

targets are.
A **very important** observation is that we are not able to provide that to our network:

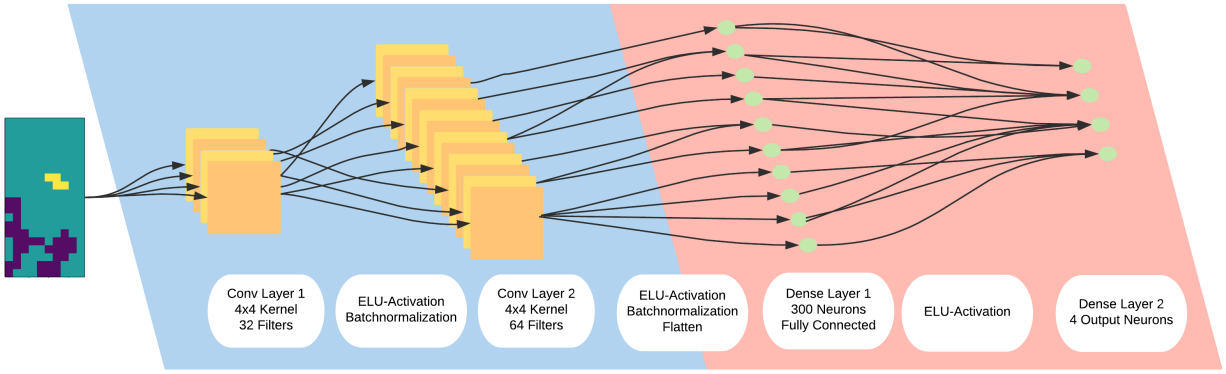
Imagine we have some state S and choose to perform action A. We are able to observe the immediate reward but in order to calculate the true total future reward, we would need to try all the combinations of actions until the terminal time n to find the maximizing sequence. A computational burden that my computer surely cannot stomach (and neither EulerCluster).
At this moment, you should really be stunned by the difficulty of this problem.
Our rescue comes in the form of the **Bellman-Equation**, giving us a recursive formula for our Q-Function:

$$Q(S_t, A_t) = r_t + \gamma * \max_{A_{t+1}} Q(S_{t+1}, A_{t+1})$$

There is an intuitive explanation for this equation.
Consider the maximal future reward at time t for action A_t. We now play action A_t and get a new state S_{t+1} and a reward r_t. The maximal future reward at time t therefore consists of the reward r_t we got for action A_t plus the maximal future reward we get now from time $t+1$ on, considering all actions. That's why we take the maximum over all actions to guarantee that we are choosing the best sequence of actions.
But how should that now help us, we have exactly the same problem again:
We don't know what $Q(S_{t+1}, A_{t+1})$ is either. . .
Now comes a very weird idea that takes some moment to sink in, at least for me:
We again use our same Neural Net to predict the value on the right-hand-side of the Bellman-Equation, after all it is the function approximator and we just need to plugin another state S_{t+1} and another action A_{t+1} right?
This somehow means that the target for our Neural Net will be produced by the Neural Net itself. Obviously we're still not getting the true targets, but at least we're getting closer by including the observed reward r_t.
This makes Reinforcement Learning so interesting, as it is a mixture between unsupervised and supervised learning.
Even drier theory has shown that asymptotically (meaning that if we play Tetris forever) we will converge to the true Q-Function. Crazy right?!

Network Design

Let's get now into more detail how our Neural Network will look. Since we want to feed our Network with raw images of the game screen, it makes perfect sense to use Convolutional Neural Networks as they have shown to be very capable of recognizing important features of images.
But hang on, we also wanted to input the action we choose, in order to get an estimate of the maximal future reward associated with this action and the current state, right? This is totally correct and was also my first thought when I read the paper of **Deepmind**.
But there is a more clever way to design the network.
Remember that we will use the **same** network to evaluate the right hand side of the Bellman-Equation. This means we need to calculate a maximum over all actions, leading to an evaluation of the network for every single one of them.
A faster and more elegant approach is to only feed our net with the current state and let it produce an output of the size of the action space, each component corresponding to the total future reward associated with a particular action. To calculate the right-hand-side we therefore only need to evaluate our network once.
In the case of Tetris, we need the output to be of size four since we have four possible actions. This means we will input only the current game screen and we will get the total future reward for all actions.
Let's summarize the architecture in a picture:



We used two Convolutional Layers of with 32 and 64 filters respectively, always using the **ELu-Activation** and **batch normalization** between the layers. We used 4x4 filter size in both layers and strides 1x1 and 2x2 respectively. We didn't preprocess the input since we have written the game ourselves, so our Tetris-Output was already fairly simple. To be honest, there is no deep idea behind the chosen parameters and the architecture, it's actually quite random. The only a bit unusual thing is that we are **not including any Pooling Layers**.
Pooling Layers help to simplify the obtained output from the convolutional layers and try to make our feature-detection translation-invariant. For image classification for instance, this means that it doesn't matter if the cat is in the top corner of the image or at the bottom left corner. This might be desirable for image classification but in case of Tetris (or learning games in general), it is rather ill-suited since the exact position of every object is needed. Our images are anyways small, so the simplifying effect of Pooling isn't needed as well.

Replay Memory

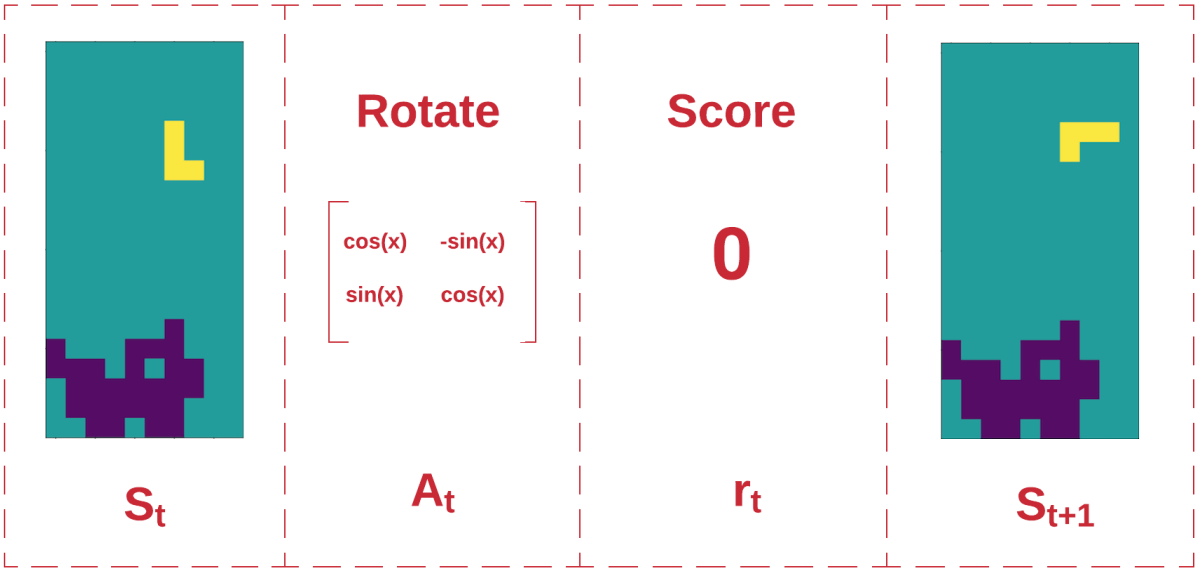
Another problem is the convergence of our approximator. Most of statistical theory relies on the **i.i.d.** assumption (independent, identically distributed) on the training samples, leading to various convergence results of Machine Learning algorithms. Of course this assumption never exactly holds in reality, but too strong violations lead to convergence problems.
Independence roughly means that each sample shouldn't have to do too much with the other samples. If we play a game of Tetris and train our network immediately in each step, we obviously can't assume independence:
Preceding states all of course look very similar as only one object has maybe moved one step. Unfortunately those are exactly what we feed into our net, which makes learning very hard and unstable.
Also, our network tends to forget its experiences, so giving it some refreshments by feeding it the same observations again can be very beneficial.
To tackle all these problems at once, **DeepMind** introduced **Replay Memory**.
Replay Memory is a storage object where we will save all the experiences of our agent. More precisely, in each step of the game, we store the following quadruple in our memory:

$$\{S_t, A_t, r_t, S_{t+1}\}$$

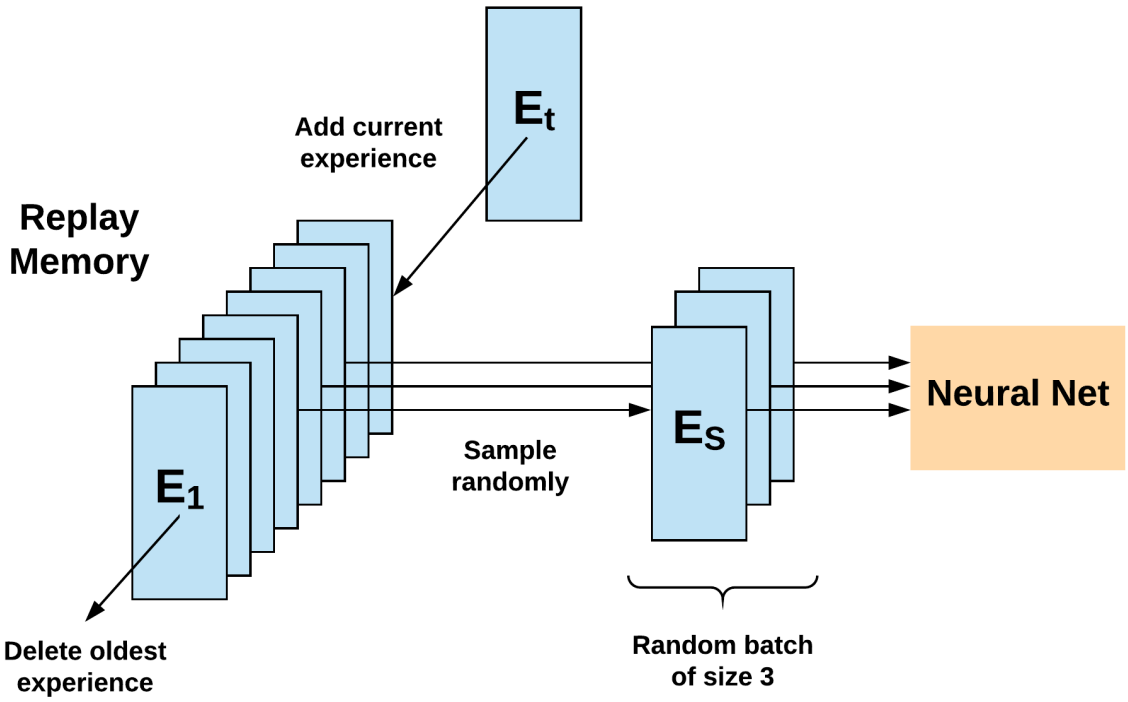
With this information, we can calculate:

1. Our **estimate** $\hat{Q} = Q(S_t, A_t)$
2. Our **target** $Q = r_t + \gamma * \max_A Q(S_{t+1}, A)$

and therefore update our network.



What is so great with this structure is that we are not forced to train our network step by step, but rather **sample randomly** from our memory. This way we're breaking the strong correlation between samples, a bit **reminiscent of bootstrapping**.
We use a fixed size for our memory, always delete the element at the last position and add the newest experience at the first position. This way we're also allowing training with the same sample **multiple times**.



To incorporate all these features, we used **Deque** from the module **Collections**, since it exactly offers the functionality needed for our memory object.

Exploration vs Exploitation

A key problem in Reinforcement Learning is the **Exploration-Exploitation Dilemma**. Suppose we had a really small state space. We would never have to worry about maybe missing some good strategies, since it is very likely that we try out every configuration of our environment. As already pointed out, our Tetris states live in $\{-1, 0, 1\}_{10\times 20}$.

Of course many of these states may be useless, e.g never encountered in a true game, still though we're never able to truly explore our whole "useful" state space. Our neural network will rapidly have some good experiences, tempting it to repeat the same strategy. This may be part of the plan, but in training, especially at the start, we prefer to try out different strategies instead of sticking to the first good one. It is kind of similar to the **Local-Global Minima problem**, where we don't want to choose immediately the first encountered Minimum of our cost function but rather check out other parts of the domain.
To achieve this in Reinforcement, we let our agent play **completely random** from time to

time.
We follow the so called **Epsilon-Greedy Learning**, where we flip a biased coin with mean ϵ, deciding if we play randomly or let the Neural Net choose the action.
This way, we allow our agent to deviate from the chosen strategy, leading to sometimes better rewards and therefore to better strategies. We anneal ϵ over episodes, making it smaller in every round because in the end our agent should learn mainly with his chosen strategy. We chose an exponentially decreasing annealing schedule as follows:

$$\epsilon_i = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) * e_{-\beta i}$$

where β is the decay rate. Note that this converges to ϵ_{end}, we're leaving the possibility to always play a bit randomly in training.