



# 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 programming it with a simple example. 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?

## The Theory of Reinforcement Learning

### Definitions

Let's first get to dry theory. How the heck 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  $t$ :

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

- **Discounted Future Reward:** The future points we get at time  $t$ :

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

- **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

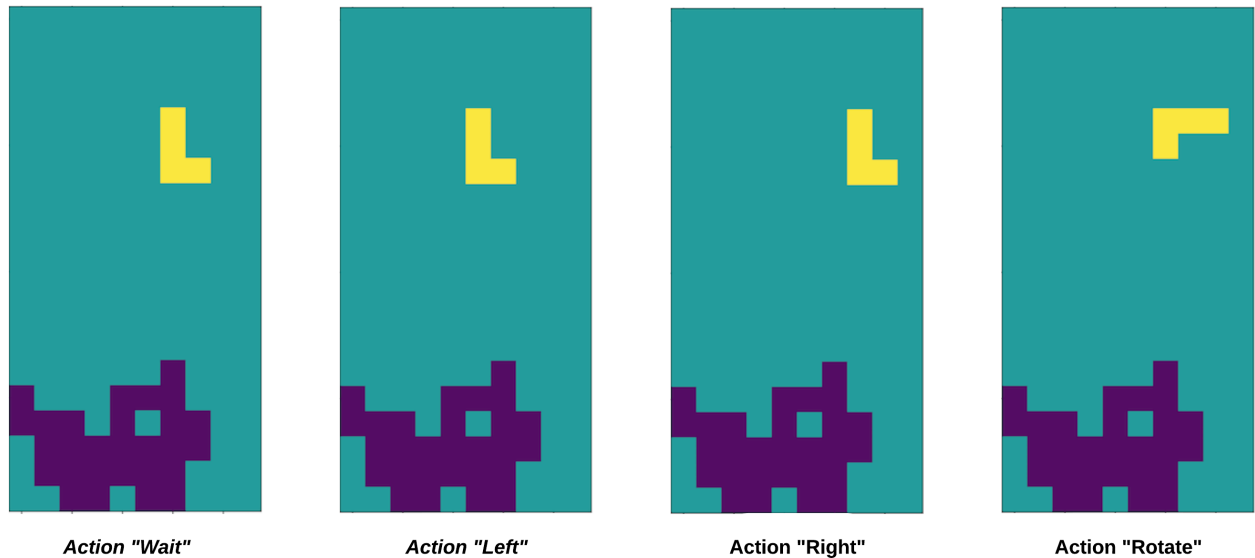
the possible actions that our agent can perform are *Wait*, *Left*, *Right* and *Rotate*.

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)



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

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

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 immense since the number of possible states is given

by  $3_{20 \times 10}$ . 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 reward, we would need to try all the combinations of actions until the terminal time n to find the maximizing sequence that gives us the true total reward. 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:

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

But then we have 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  $r_t$ .

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? This is totally correct and was also my first thought when I read the paper of **Deepmind**.

But there is more clever way of designing 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:

