Advance machine learning (Fall 2023)

Hossein Yahyaei
Mehrdad Baradaran
Katayon Kobraei

Abstract

In this work we are dealing with delayed reward functions in the Tetris environment which is known to be complex to apply the machine learning technology such as Q-learning as well as how a deep Q agent can play in such a big environment.Such games as Tetris game is known to be a delayed reward system because of its characteristics of generating sparse reward in learning process.Also, since the number of cases is very large due to the various block types and order It's hard to make sure that the agent can explore variety of cases to learn playing the game. So we are going to take a look at challenges and how we overcome these challenges.

Introduction

Recently, artificial intelligence has applied to many different fields of applications and the various research have led to a better learning performance. Therefore, it is a great challenge for those who study those fields which have more irregular and complex characteristics of the problems.A various types of approaches and techniques are being studied, and among them, the most notable one is reinforcement learning, which recognizes the current state at every moment and autonomously makes optimal decisions. Reinforcement learning is an algorithm that defines the environment of a problem to be solved as a state, an action, and an expected reward so as to make autonomous decision making in the direction to maximize the value expected in the future This decision-making structure is expected to solve the complicated problem of choosing the optimal policy in a sudden change of situation because it recognizes the state of data and makes the decision for each situation unlike the other learning methods such as map learning.

However, reinforcement learning when it is applied to the real environment shows many problems and limitations. In certain industries or environments, unexpected situations arise in many circumstances, or high-dimensional data that are not able to be inferred or learned is often utilized for better decisions. In these unexpected and irregular environments, reinforcement learning shows clear limitations and various factors that negatively affect the learning performance. Delayed reward system such as the Tetris game is one of the applications

where we can not use the conventional value functions based on state-action pairs since the expected return has to be predicted from every state-action pair . A single prediction error might hamper learning. Also to mention this delayed and sparse reward leads to the need of a large number of episodes for an agent to clearly grasp almost all parts of the environment.

Experiment

At this point It's better to mention what kind of agent we use ? At the beginning of the project we used the normal DQN without any extra features or training mechanism , But this step was not really beneficial because as we all know DQN without a prioritized replay buffer is not really effective in such a big environment with sparsity of its rewards. The agent neural network consisted of 2 convolution layers with zero padding and also a linear classifier. This simplicity is because we wanted to evaluate how good our environment is and then create a bigger and more complex agent . In this part we will explain each and every step that we've been through for playing the Tetris game using a DQN based agent. These are the main steps:

- **Game dynamics** : how does the original game dynamic works
- **Environment** : how do we implement the environment and its reward system and how it affects agent learning .
- **Agent** : what is our agent and all of its details

Game Dynamics

The dynamics of the Tetris game make it an intriguing and challenging environment for reinforcement learning agents. Tetris, a popular tile-matching puzzle game, requires players to strategically place falling blocks to create complete horizontal lines. As the game progresses, the speed of the falling blocks increases, demanding quick decision-making and adaptability.In Tetris, the game board consists of a rectangular grid in which the blocks descend from the top. The blocks come in seven shapes, known as tetrominoes, which are composed of four squares arranged in various configurations. The player's goal is to rotate and move these tetrominoes into 3 directions that are left, right and down to form complete horizontal lines without leaving any gaps to eliminate that line from the game grid ,Thus based on this knowledge we knew that our actions are going to the left , right ,down or rotate but there is one more action for the player which is doing nothing and letting the block land by itself. The dynamics of the Tetris game are characterized by the continuous descent of the blocks, the limited time for decision-making, and the need to consider both immediate rewards and long-term consequences. This dynamic nature poses a significant challenge for reinforcement learning agents, as they must learn to make optimal decisions in real-time while accounting for the constantly changing game state.

Environment

We first implemented the game itself by using **pygame** library and played it to grasp its dynamics and then the application was converted to the environment by adding a reward modulation system. Then, to add this reward system we added a couple of functions. However,

in this journey we faced different issues which made us ignore those functions/methods we'd implemented.In this custom Tetris environment that we implemented, the first thing to mention is what do we feed the network with as the input ? State is in our case equal to the observation which a human kind can have by looking at the game board that is rendered RGB image returned by the pygame render function. The most important problem that we faced in this work was enormous computation complexity and time complexity which we couldn't overcome because we had no good enough computational power as bachelor students.

Structure of agent interaction with the environment:

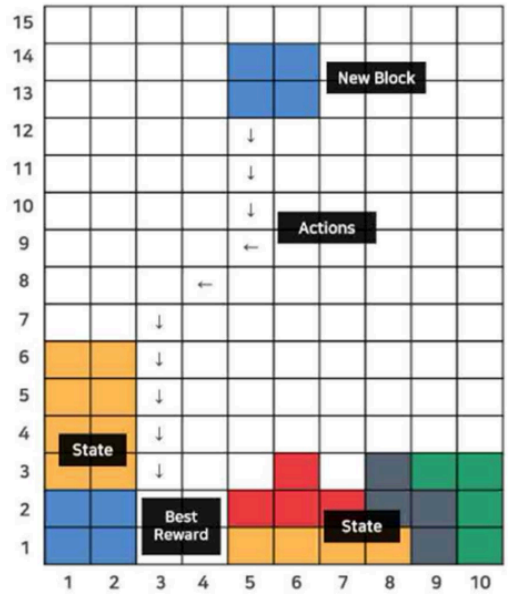
1. *Initial state is returned by the environment to the current state .*
2. *While loop until agent lose:*
    a. *Agent choose an action based on the current state*
    b. *Environment applies the chosen action and returns the reward and also renders the new state of the game .*
    c. *Based on the new state environment defines whether an agent lost or not.*
    d. *New state sat to be the current state.*

It's time to go through each and every step:

1. **Level 1 :** In this environment which is the first environment we implemented for any time step in which the agent would pass a negative reward was considered and for every action done by the agent another small negative reward applied .But, if the agent fills a line a large positive number is applied as a reward . Based on training the simple DQN which we had in this environment failed us to improve in a small number of episodes which in our case was 100 episodes with 90 iterations in each episode and with more exploration of ours we found out that negative reward forces the agent to learn to lose faster to avoid bigger negative reward in up coming future.To do so agent had done nothing except forcing the tetromino blocks to land as fast possible to get highest negative reward. (Failure)

2. **Level 2:** In this environment we have deleted the negative reward in each step and instead a positive 1 reward is added in each step to force the model to live longer and not to lose too fast and the goal is to maximize these positive ones. But we still have this negative reward for each action that is taken by the agent that caused the agent to not act and not get a negative reward so all Tetrominos landed at the center of the board which wasn't preferable that made us remove this action's reward. Now this environment proved to learn and converge in many papers before 2020 (etc) but it was not what we were looking for because in a small number of episodes we could not visualize its convergence except a random reward which an agent could get that doesn't depend on our work so we ignore this kind of information. Thus , at this point our problem is slow convergence speed which these papers we are considering suggested 2000 to 3000 episodes but with our computational power it was not possible so we look forward to getting better results out of these environments .

3. **Level 3:** At this point we observed the slow convergence so to tackle it we came up with a better reward function which is explained down below and sat to be our final environment reward system .



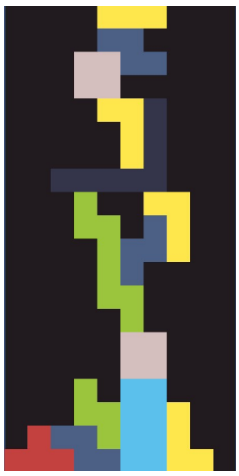
In this environment we have divided the reward into two main types :

1. **Reward before landing the Tetromino**
2. **Reward after landing the Tetromino**

Usually we would look at these two rewards as they are the same but they are not. This was what we have mistaken all this time.So by differentiating these two a new straight forward approach could be applied.
Now it's better to explain this differentiation . When we are looking at the Tetris game we see that if a Tetromino block is landed correctly a great positive reward would be given but what should be given to the agent as a reward while the Tetromino blocks are moving (not landing yet)? This reward is known as RAL *(Reward after landing the Tetromino)* in this report .

So first we discuss Reward system after landing Tetrominos ,It consist of three different rewarding sections:

- **Game over** : This is set be -30 if at least one block hits the top of the game board just like the picture below .



The way we choose to use -30 is by trial and error that shows us this number is better with respect to other defined rewards which we will look at.

- **Score** : Score is the reward that is defined by the number of lines that is created after landing the Tetromino . It means if the agent lands a Tetromino that makes two filled lines, the score is calculated as the number of filled lines times 100.Just like down below if a complete like is created +100 reward is given by the environment and this line is eliminated from the game .

- **Joints** : This reward is defined as a positive reward corresponding to the number of paired blocks that are joined together in a line horizontally after a Tetromino block is landed. This means until there is no landing for a Tetromino block there would be no returned reward from this reward function . For each every number of joints we regarded a unique and increasing static reward to differentiate between 5 blocks creating 4 joints and 4 blocks creating 3 joints. Below table shows different values for different numbers of joints . In this reward system as its formulation shows down below *(this formula is for the time that the number of joints are bigger than zero)* we consider the distance of the landed block from the bottom of the board to increase the reward if it is closer to the bottom of the game board.

Number of joints	2	3	4	5	6	7	8
Joint reward	1.01	1.04	1.08	1.1	1.4	1.5	1.7

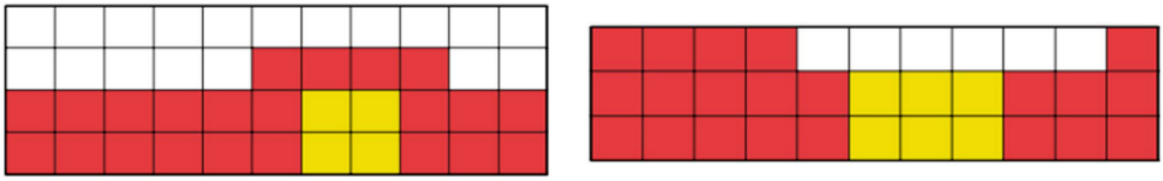
$$\left(\frac{1}{(FieldHigh - row)}\right)\times\left(JointReward_i\right)^{(number\ of\ joints)}$$

In the above formula JointReward_i shows the number of joints which have been created so far. This reward shows that even an uncompleted line has a great reward if it lands next to other blocks and makes no gaps in between.

In the second part of the environment we will look at reward systems that are associated with those states that are before a Tetromino lands . In this part the only thing I could've applied was to consider empty spaces that are made in the game board to show the agent that it could have filled the maximum remaining empty blocks but it didn't do so because it hasn't done any action to land the Tetromino in the right spot.This is how we forced the agent to act faster to overcome faster decision making this idea of reward in each steps before Tetromino lands is proposed in 2020 paper of enhancing reinforcement learning performance in delayed reward system but we improved over this idea and tried the exponential rewarding mechanism as the number of empty spaces increase . Because we are focusing on making the reward grow exponentially in case of determining difference among different numbers of empty blocks We used the below formulation for the reward.

$$-1\times\left(\left|EmptyBlocksReward^{NumEmptyBlocks}\right|\right)$$

By empty blocks there might be two scenarios that occur: unreachable empty blocks and reachable empty blocks. We did not differentiate these two kinds but you can see both of these scenarios in the pictures below.In these pictures yellow blocks define the empty spaces that the negative reward is associated with.



Finally we kept the positive 1 reward as the agent is being alive in the environment and the sum of all of these different rewards defined our final reward for each action at each step.

Agent

In this section we'll identify different types of agents we've implemented and tried on this environment so you can check them in the attached files.

Neural networks:

- Simple network : As its name shows it is the simple network consisting of two convolution layers with padding and a single linear layer as classifier *(It does not have softmax, do not misunderstand it)* . But this architecture for this task is not suitable as it can differentiate between states but it is not able to differentiate among different actions but it is good enough for evaluation .
- Clip based network:This network uses two convolution layers plus a linear layer to embed the current state of the environment and then concurrently 5 different numbers associated with 5 different classes or in other words 5 different actions are embedded using nn.Embedding with a non linear layer to embed them in the same dimensions as the state of the environment. Afterward each vector associated with each action dot product with a state vector to define the Q value of that specific action associated with the current state .Thus, we have 5 different values which show the Q value of each action at a given state. This architecture which is proposed in Learning Transferable Visual Models From Natural Language Supervision article as the basis idea is more powerful and insightful than it looks because when each action gets encoded and embedded separately they are able to differentiate between the consequences which might occur in future as it tries to model this reward system itself.

Agents :

- DQN:

For the agent we have used Deep Q learning in its blank and normal form to evaluate our environment and the network we have used in this environment is the Simple network that is explained earlier . This agent can not do much for many reasons , first one is it lacks stability and burst in Q values these are the reasons to use double version of DQN and second one is the problem of sampling experiments for its training because all samples have same probability to

be chosen for training process it lacks faster convergence and accurate convergence because those samples that force great loss to the network are seen by the network as much as those samples that were not inaccurate choice or not wrong choice to be made by the agent .
This agent used linear epsilon decay for its training and the result was not really good but it showed us that as the environment gets large exploration is needed. So to tackle this exploration problem we've used reward based epsilon decay proposed in RBED paper as a solution but again the problem of computational power prevents us from running such large training for too long.

- Double DQN :
  Double DQN used to overcome stability and burst of Q values of the DQN's to get much better and for training this agent we also used linear epsilon decay but we implemented RBED as well if we could enough computational power we would run to see the result of using RBED . This RBED helps to explore the game space as much as the agent needs because if the agent does not improve over its reward by having its previous experiences there is no need to exploit thus the agent needs to explore more. Below image shows the algorithm proposed in RBED paper .

Algorithm 1 ε-decay and reward threshold increment step

if $last\_reward >= reward\_threshold$ **then**
  $\varepsilon \leftarrow decay(\varepsilon)$
  $reward\_threshold \leftarrow increment(reward\_threshold)$
end if

  Double DQN used in this work is exactly the same as DDQN in its paper .Below algorithm defines how we used DDQN .keep this in mind target network update frequency sat to be 1/512 this means after 512 iteration target network is updated using online network parameters.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

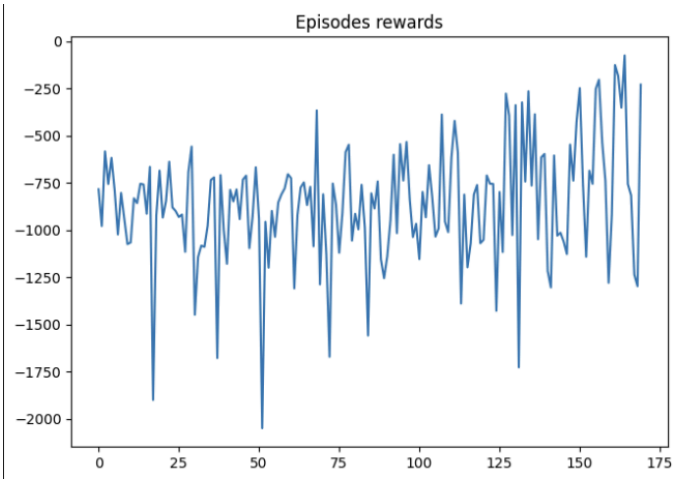
  We did not use any pretrained model for our agent or even two different networks that are differed in the architecture but people can improve our work by doing these as our future work.
- DQN with Prioritized replay buffer :
  As shown in the algorithm below, we used an improvement over DQNs known as prioritized replay buffers to force the neural network first not to forget what it learned in

the past. With use of its performance on each state, action, reward and next state that is stored as a sample in the buffer to indicate how far (how bad) the agent was from the correct (suitable) action in each of these samples to prioritize their sampling at each iteration of learning done by the agent.However,this prioritization forces the network to not miss its performance on any of these samples it's like forcefully pushing the agent to learn every single sample that it had seen through its interaction with environment . This approach of prioritization exists before the prioritized replay buffer, known as weighted least square but with slightly different implementation which we could have used that one as well in this scheme (people can do this as future work) .

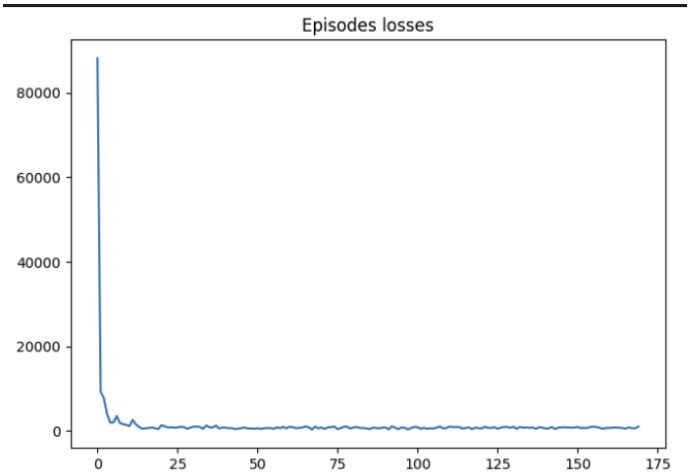
Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

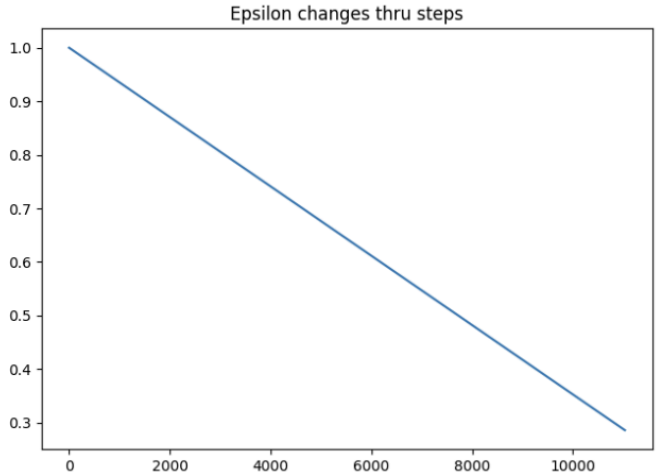
This agent got reasonable results as it is shown down below. This agent learned to increase its incoming reward but unfortunately because of lack of computational power to run such an agent for 1000 episodes we are not able to show the final best result that our agent is capable of getting . Thus if you are interested in the final result and you have the computational power to run this for 1000 episodes you are allowed to do so.



Below plot shows the convergence of the model by decay in loss function ,and again this shows how good the agent is able to converge.



And finally the linear epsilon decay plot is shown down below and this plot is for the time we didn't use RBED and it is important to mention using RBED in our code is as simple as changing a variable to set the epsilon decay to be reward base.



This type of code is also applied to transformation from DQN to DDQN as well so that people can change a single variable to use DDQN instead of DQN .

- Double DQN with Prioritized replay buffer
  This agent is the same in nature as the DDQN but with a slight difference which is its replay buffer that is replaced by its prioritized form.

We could have added extra information about our work but this information is not essential in someone's journey in improving their agents to learn Tetris game.