An Implementation of the Deep Q-Learning Algorithm with Tetris

Emilio Kiryakos
UIC College of Engineering
University of Illinois at Chicago
Chicago, USA
ekirya2@uic.edu

Dev Patel
UIC College of Engineering
University of Illinois at Chicago
Chicago, USA
dpate366@uic.edu

Markus Perez
UIC College of Engineering
University of Illinois at Chicago
Chicago, USA
mgperez2@uic.edu

Michael Klimek
UIC College of Engineering
University of Illinois at Chicago
Chicago, USA
mklim7@uic.edu

Christopher Bejar
UIC College of Engineering
University of Illinois at Chicago
Chicago, USA
cbeja2@uic.edu

***Abstract*—Our project had the goals of train multiple agents to play the game of Tetris using Deep Q-Learning in order to 1) evaluate the efficacy of different reward functions for Tetris and 2) build, to our knowledge, the first open-source dataset of Tetris Gymnasium observations (current state, action, reward, next state) for possible use by others in future offline reinforcement learning Tetris projects with a fall back goal of a ”good” agent**

I. Introduction

Tetris is a popular puzzle game developed by Alexey Pajitnov in 1984, which has players solve the game by stacking block objects in order to clear lines for points, all while intensifying the pace of the gameplay as the game progresses.

The game of Tetris has a variety of block shapes and colors with line, square, T-shape, l-shape and zig-zag blocks, all also called ”tetrominoes”. The game is scored based on how many lines were cleared at one time and how fast were the lines were cleared. Players of the game have to strategically think about the placement and rotation of Tetris blocks to be able to maximize the number of lines cleared at once and manage the blocks the player holds and has stored in the background of the game.

As the game progresses, the player places the blocks from bottom to top and the game is over when the any part of a block reaches the top of the stack, also called the ”Skyline”.

Reinforcement learning is one of the three major machine learning paradigm that focuses on the study of intelligent agent behavior with the goal of reward maximization, where rewards are scores accumulated from a reward function that rewards different actions by an agent within an environment.

Q-learning is a model-free reinforcement learning algorithm first described by Chris Watkins in 1989. Deep Q-learning is an extension of Q-learning utilizing deep learning that was first shown by Google DeepMind in 2013.

II. Problem Definition

The design of the reward functions is crucial for efficient and adequate training of agents. With a game as complex as Tetris, there are numerous considerations one must consider. To what extent should we reward individual line clears? How can we properly incentivize going for clears of four rows at a time? Should we attempt to target the behavior of average humans? Professional players?

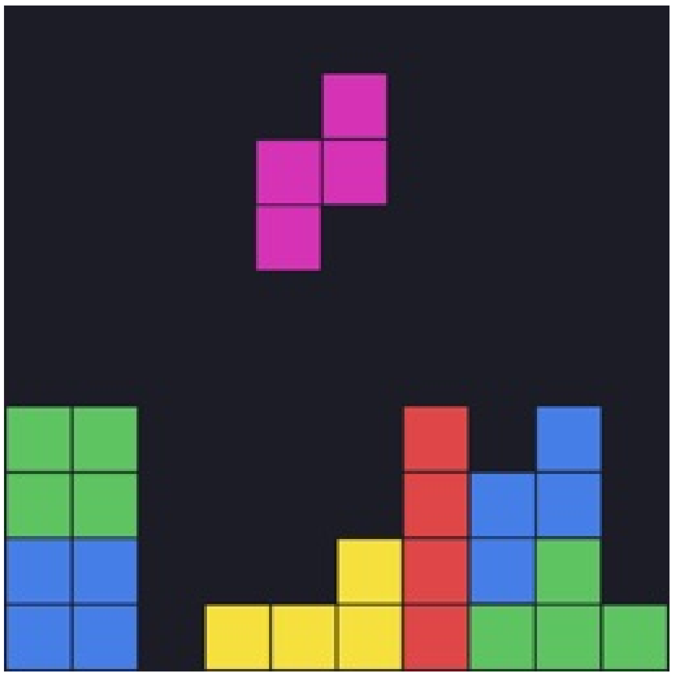
Fig. 1. Simplified capture of typical Tetris gameplay

With no clear ”correct” reward function, we primarily aimed to benchmark and compare different informed reward function designs. Our current implementation was done through a large series of trial and error, testing what did and didn’t work.

One of our goals was to create something other people would be able to use for their own projects, a dataset of [state, action, reward, next state] data from a trained agent. Offline Reinforcement Learning, defined by a lack of agent and environment interaction, requires extremely large datasets of replay experience from trained agents, of which none exist for Tetris, so by contributing in this way, we'd hoped assist others with their modern research in this field.

Ultimately, due to numerous struggles, we readjusted our target to our fallback goal of a "good" agent. We more concretely defined this goal as an agent able to reach a score of 10,000 within 30,000 episodes.

III. Data and Experimental Design

To understand our experimental design, it is important to understand the algorithm we implemented, Deep Q-Learning, which itself is an extension of Q-Learning.

A. *Q-Learning*

In reinforcement learning, agents are in an environment where at each time step t, the environment is in a certain state s_t and the agent must choose an action a_t to perform, which rewards the agent with a predefined reward r_t. Q-learning is a classic reinforcement learning algorithm with the objective of learning an optimal policy π^* that maximizes expected reward. The q-value, a measure of "goodness", for the optimal policy is given by the Bellman equation:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[r_t + \gamma \max_{a'} Q(s',a')]$$

where γ is the discount rate of future rewards. The equation essentially states that under the optimal policy, the q-value of any state-action pair is equivalent to the expectation of the immediate reward and maximum discounted reward of the next state.

The algorithm, detailed below, has proven itself useful throughout the history of reinforcement learning but possess major limitations. Namely, the size of the q-table, table of q-values for each state-action combination, and the required discrete nature of the state and action spaces.

Algorithm 1 Q-Learning

```
Initialize $\hat{Q}(s,a) = 0 \forall s,a$
Observe initial state $s = s_0$
repeat
    (1) Choose action $a$
    (2) Observe reward $r$ and new state $s'$
    (3) Update the q-table $\hat{Q}$:
        $\hat{Q}(s,a) = (1-\alpha)\hat{Q} + \alpha[r + \gamma \max_{a'} Q(s',a')]$
until convergence
Optimal policy: $\pi^*(s) = \max_{a'} Q(s',a')$
```

B. *Deep Q-Learning*

The Deep Q-Learning algorithm, the algorithm we implemented, address the limitations of the tradition Q-Learning algorithm by taking advantage of deep neural networks, hence the name. The key components of the Deep Q-Learning algorithm are as follows:

- **Functional approximation of Q-table**: The algorithm utilizes a neural network, a "Deep Q-Network" (DQN) or "policy network", which takes in state tensors and outputs Q-values for each possible action. This design also requires that there be a second neural network, a "target network" of the same architecture, with a "lag" on its parameters in order to prevent temporal correlations as a functional approximation of the Q-table where the maximum future Q-value is computed
- **ϵ-Greedy algorithm**: In order to balance both exploration and exploitation, the next action an agent takes is determined by an ϵ-greedy algorithm. This means that the agent will take the optimal action, the one with the highest q-value, with $1-\epsilon$ probability and a random action with ϵ probability
- **Experience Replay**: In this algorithm, the training and experience gaining phases are intertwined but not happening at the same time. While the agent interacts with the environment, we store [state, action, reward, next state] data in a table. The learning phase occurs when random samples are taken from that table to train the policy network. This provides the advantage of using experience multiple times, allowing us to get the most of it despite the low learning rates, and better convergence rates as the data is "close" to being i.i.d.

C. *Data and Experimental Design*

We implemented the Deep Q-Learning algorithm with a few metric captures in order to assess agent performance. We mainly tracked episode rewards, cumulative reward, best reward, and lines cleared per episode. These metrics when graphed gave us clear indications of how our code was working. We wanted to see episode rewards go up, an exponential cumulative reward graph, a consistently increasing best rewards, and an exponential lines cleared per episode.

These metrics allowed us to compare different network architectures, reward functions, and combinations of the countless parameters.

IV. Results and Analysis

A. *Overall Results*

Ultimately, we were able to achieve an impressive result in training our agent. The best result our agent was able to complete was 8,160 lines cleared and 62,300 points. This high of a result shows concrete ability to play Tetris and a strong algorithm capable of comprehending the nuanced skills and future game-play planning required for Tetris. Although 62,300 was our highest overall instance, the model consistently performed at a result receiving roughly 16,000 points and 650 lines cleared every game. A significantly lower number than its high score, but still an impressive result which shows beyond average human competency and successful training.

The agent began with an abysmal result, finishing instances in only a few seconds as it was not capable yet of using curated strategy to clear lines and increase the duration of the instance. During early training, the model consistently received point results in the range of 20-40 points. Throughout training the performance followed an exponential pattern, remaining consistently low before suddenly having a rapid improvement in performance. This can be seen in figure 2 and figure 3 as the performance indexes have a large jump later in their training to where they begin to achieve impressive results.

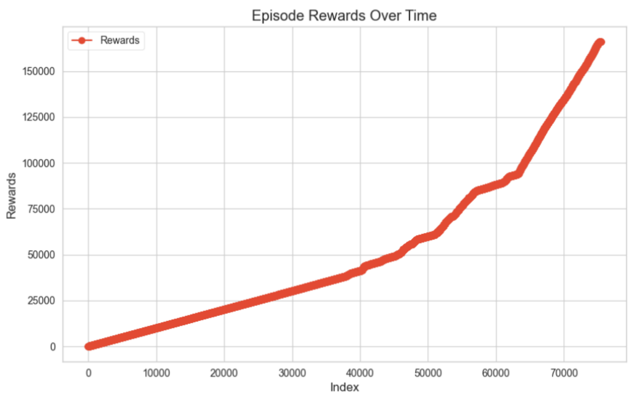

Fig. 2. Cumulative Points Through Training Duration

The curve in figure 2 shows that the training is a success and the agent is improving in ability. Had the agent been unsuccessful, the line would be completely linear (as it is in the beginning when it struggles in game play) but its exponential curve shows improvement since it means point totals in each game increase.

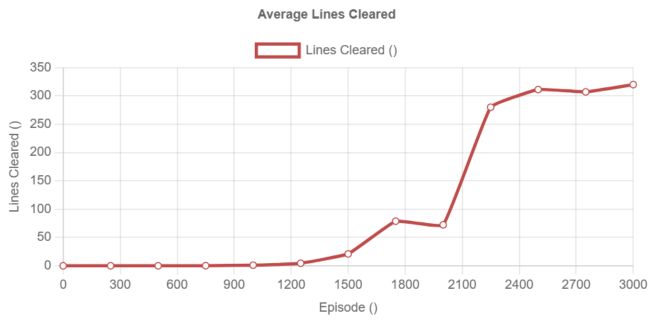

Fig. 3. Running Average of Lines Cleared

Figure 3 shows a similar story as Figure 2. The graph shows a running average of the amount of lines cleared during every Tetris instance. As the training continues through time, the pattern of the average follows a similar one as the rewards over time, with the initial few hundred episodes showing little progress but a large jump and improvement following. Again showing an exponential pattern, although with a plateau, and overall successful training as a good average of lines cleared is reached. This result is ideal, as the plateau shows that we were able to create a model, that still with occasional drops and instability, is able to play at a consistently competent skill level.

B. *Training Experimentation and Analysis*

To achieve this successful result, we had to go through extensive trials training the agent. Every trial and attempt included changes in one or more hyper-parameters looking for an optimal best training model. These parameters included learning-rate, epsilon-decay rate, batch-size, number of episodes, and the buffer size.

Because of the complex nature of Tetris, as it has elements requiring planning for future strategy and a large number of possible actions, tuning all these parameters proved to be particularly difficult. For many attempts, we were stuck with little progress and unable to create a result that made any progress in game play. After finding a combination that was able to successfully train, it remained difficult to tune it to improve that training significantly. Ultimately we were able to find a working and fairly-well optimized combination of the following parameters:

- Learning-rate: 0.0005
- Batch-size: 32
- Episodes: 4000
- Epsilon-decay: 0.005
- Buffer-Size: 2000 episodes

This combination worked for a Tetris training model for several various reasons. The low learning-rate meant that training had to take a large amount of episodes to train, but it allowed for the agent to properly progress towards an optimal playing style without over-fitting the model or missing correct weights and biases. The low batch-size allowed the model to adapt more quickly to changes in its performance and account for them in its training, with the added benefit of also improving the speed of training. The number of episodes training and the training-buffer size went on for long enough for the model to gain enough information from its game play without going too long and over-fitting the model.

The parameter we did the most tuning of was our epsilon-decay. A greedy-epsilon algorithm is one in which an action is taken based on either choosing a random action (exploration) or one based on trained Q-values (exploitation) with the latter becoming more frequent as the training progresses. The epsilon-decay is the factor that decides how fast the training should progress from exploring to exploiting. After testing several different patterns of epsilon-decay like an exponential decay and a quadratic function, we found that a linear function had the most success with its simplicity, as the other functions allowed for too much exploring and could not create a successful agent. There is a possibility that training done with a larger batch size could work with a larger amount of exploration but we were unable to find it through our experimentation.

Figure 4 shows our training's epsilon decay over episodes of training, a linear pattern is present until it hits the floor of an epsilon of 0.01 (1% chance of exploring, 99% chance of exploiting).

C. *Training Challenges and Analysis*

1. Unstable Training: A significant challenge which we encountered throughout training and have only been able to

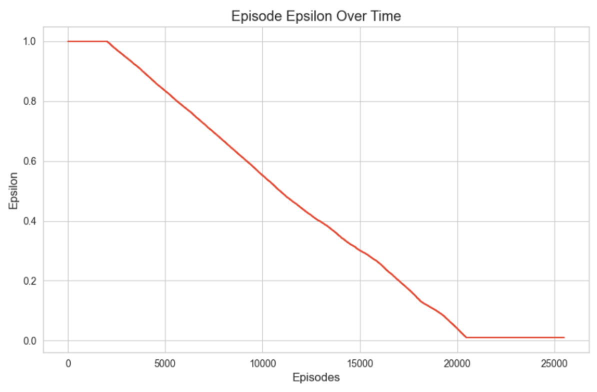

Fig. 4. Epsilon Decay Pattern

partially resolve is instability. Through training, our agent improves over time but still faces challenges with losing its progress and returning to an unsuccessful strategy before relearning and being able to play optimally again. Our first iteration of successful training displayed this phenomenon heavily, but after tuning the hyper-parameters we were able to create a more consistent trend of improvement that better held onto proven and successful playing strategies.

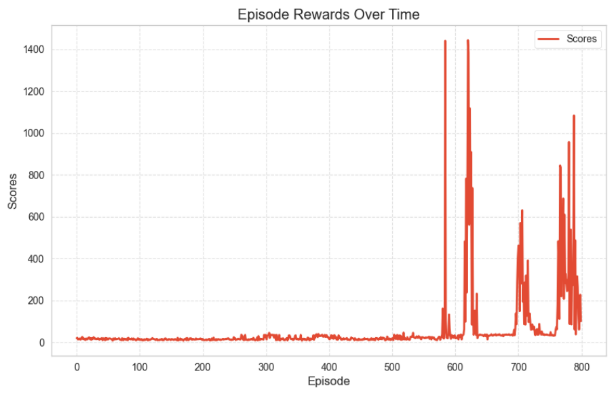

Fig. 5. Individual Episode Results Show Unstable Results

Figure 5 shows individual episode results and displays the result of several drastic increases before returning back to the ineffective strategies of before. We were able to omit this issue from our final model by stopping training early once the model was able to play successfully.

Ideally, the hyper-parameters could be further tuned to prevent instability and instead have a more consistent improvement with less drops in progress. One probable solution is adjustments to a combination of the learning-rate and batch-size. The learning-rate could be too high leading to the agent missing the lowest level of loss and instead landing in an ineffective area during training iterations, but a lower learning-rate would also require tuning of the episode amount to insure the model has enough time to train to the full extent. Increasing batch-size could lead to better results by allowing the agent to process more training data and learn from more instances of varying strategy, although also increasing the training time substantially.

2. Environment and Reward System Issues: Another significant challenge we faced was properly adapting the model to its environment and experimenting with a reward system.

We initially adapted our agent in the Tetris Gymnasium environment and attempted to build a further developed reward system that more highly rewarded things like lines cleared, longer duration, and smaller gaps, while penalizing short durations, game terminations, and large block-stack heights. We experimented with various functions within implementing the reward systems as well to increase them based on severity. Ultimately, we were unable to create a combination which properly allowed the model to train.

After numerous unsuccessful trials we adapted our same model to an open-source Tetris game written in Python with PyGame. After this change we found much greater success in training, proving that it was not our model that was ineffective but instead either the Tetris Gymnasium environment and/or the reward system. Once we switched we were able to focus more or on the training aspect and tuning the hyper-parameters.

V. Remarks and Future Works

What worked well in our project was our back up goal of getting the agent to be able to learn how to play the game of Tetris to a ”good” level.

We faced challenges from the get go, starting with needing to find a way to use the SSH from the GCP (Google Cloud Platform) virtual machine that we created to connect with our IDEs.

These continued through model development, from the environment issues we described to our unfamiliarity with PyTorch to poor results which required a lot of work to find the source of.

Ultimately, we are proud of the result we ended up with, a well-implemented algorithm from network architecture to training loop to resulting agent.

We could improve our project by revisiting our original goals of a comprehensive study and dataset, exploring additional alternative network architectures, more extensive hyper-parameter tuning, and incorporation of more advanced RL techniques like a double DQN or a dueling DQN.

VI. Novelty Statement

A. *What We Did Differently*

During the design of the reward function in our Tetris project, we used a systematic process that involves:

1) **Advanced Reward Composition**: We have experimented with multi-component reward function instead of simple line clearing rewards by focusing on:
   - Exponential scaling feature for the bonuses
   - Application of penalties because of stack height
   - More Penalties for gaps

- Implementation of final state penalty to avoid instant game endings

This came to mixed results, we found that the reward structure had large implications on over-fitting and preventing successful training. We can conclude and ideal training model would focus on line-clearing rewards with additional but smaller penalties and rewards on other factors.

2) **Dynamic Reward Scaling**: We experimented with a nonlinear design:
   - Applying an exponential function for rewards
   - Continuously applying penalties on stack height and difficulty level of board
3) **Environment Experimentation**: As we know Tetris Gymnasium environment provides a flexible and standardized platform for reinforcement learning experiments, so we experimented with using the environment and developing reward systems complimenting it in efforts to train the model faster and better. We found that the environment was limited and that finding success in training was difficult. When we replicated our model using an open-source Python Tetris game we found better results, showing that the gymnasium environment needs further exploration and experimentation.
4) **Unique Approach**: Our main contributions and conclusions from our experiences are showing that the reward function has just as large of an impact on training as more traditional hyper-parameters and that in a game-setting, learning-rate and batch-size are the two most impactful hyper-parameters in successful development.

VII. Contributions

1) **Chris:**
   a) **Graphing and Analyzing of Metrics**
   b) **Assisting with Code Review/Debugging**
   c) **Slides 2-6**
   d) **Project Proposal Literature Review and References**
   e) **Project Report Problem Definition**
2) **Michael:**
   a) **Implementing ϵ-Greedy Algorithm**
   b) **Monitoring training**
   c) **Tuning Hyper-Parameters**
   d) **Experimenting with environments and neural network architectures**
   e) **Slides 11-14**
   f) **Project Proposal Data**
   g) **Project Report Results and Analysis**
3) **Markus:**
   a) **Implementing Deep Q-Network**
   b) **Implementing Target Network and training loop**
   c) **Designing reward functions**
   d) **Implementing Model Saving and Refactoring**
   e) **Slides 7-9**
   f) **Project Proposal Introduction, Problem Statement and Objective, Data, Methods and Feasibility, and Time and Team Assignment**
   g) **Project Report Data and Experimental Design**
4) **Dev:**
   a) **Identifying, paying for, and setting up our GCP VM**
   b) **Implementing Experience Replay**
   c) **Project Proposal Time and Team Assignment**
5) **Emilio:**
   a) **Initial Environment Setup**
   b) **Agent Training**
   c) **Slides 10 and 15-16**
   d) **Project Proposal Introduction, Problem Statement and Objective, and References**
   e) **Project Report Remarks and Future Works and Novelty Statement**

References

[1] szityu01, “Reinforcement learning agent plays Tetris,” YouTube, May 28, 2008. https://www.youtube.com/watch?v=Uf5d3TwiiqI

[2] N. Lundgaard and B. Mckee, “Reinforcement Learning and Neural Networks for Tetris.” Available: https://www.idi.ntnu.no/emner/it3105/materials/neural/lundgaard-tetris-ann.pdf

[3] M. Stevens and S. Pradhan, “Playing Tetris with Deep Reinforcement Learning.” Available: http://vision.stanford.edu/teaching/cs231n/reports/2016/pdfs/121_Report.pdf

[4] Pygame, ”Pygame Front Page.” Available: https://www.pygame.org/docs/

[5] Farama Foundation, ”Gymnasium Documentation.” Available: https://gymnasium.farama.org/

[6] Max-We, ”Tetris-Gymnasium.” Available: https://github.com/Max-We/Tetris-Gymnasium?tab=readme-ov-file

[7] PyTorch, ”PyTorch documentation.” Available: https://pytorch.org/docs/stable/index.html

[8] Keras, ”Keras 3 API documentation.” Available: https://keras.io/api/

[9] Tetris, “About Tetris®”, Tetris, 1985-2024. Available: https://tetris.com/about-us.

[10] C. John and C. H. Watkins ”Learning from Delayed Rewards.” Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[11] Volodymyr Mnih et al., ”Playing Atari with Deep Reinforcement Learning.” Available: https://arxiv.org/pdf/1312.5602

[12] R. Agarwal, D. Schuurmans and M. Norouzi, ”Reinforcement Learning and Neural Networks for Tetris.” Available: https://arxiv.org/pdf/1907.04543v4

[13] K. Chanda, “Q-Learning,” GeeksforGeeks, Feb. 06, 2019. Available: https://www.geeksforgeeks.org/q-learning-in-python/

[14] V. Nguyen, “Deep Q-learning for playing Tetris” GitHub, Apr. 03, 2023. Available: https://github.com/vietnh1009/Tetris-deep-Q-learning-pytorch

[15] A. Gupta, “Deep Q-Learning,” GeeksforGeeks, Jun. 13, 2019. Available: https://www.geeksforgeeks.org/deep-q-learning/

[16] E. Kiryakos et al., “Deep Reinforcement Learning,” Nov. 11, 2024. Available: https://docs.google.com/presentation/d/1FK4SqN7YNi_wZYL8ip2F1ZOp6y_GmkT0RLZKjwVI3iU/edit#slide=id.g31b1747c0b9_0_50

[17] N. Slater, an answer to ”What is ”experience replay” and what are its benefits?,” Jul. 19, 2017. Available: https://datascience.stackexchange.com/questions/20535/what-is-experience-replay-and-what-are-its-benefits

[18] N. Prasad and G. Gundersen, ”Introduction to Q-learning,” Oct. 19, 2017. Available: https://csml.princeton.edu/sites/g/files/toruqf911/files/resource-links/q_learning_notes.pdf

[19] E. Kiryakos et al., ”Introduction to Q-learning,” Oct. 10, 2024. Available: https://github.com/notcoose/Tetris-Deep-Q-Learning