Mastering Tetris: AI Agent Training for Complex Decision-Making

By Ben T Boben and Rohan Joseph Jayasekara

1. Introduction and Problem Statement

Tetris is a dynamic and complex game that requires long term planning with immediate decisions. These decisions include minimizing gaps and height of the tetrominoes while maximizing the number of rows cleared. For our project, we explored three approaches to training a Tetris agent, Deep Q-Networks (DQN), Genetic Algorithm (GA), and Proximal Policy Optimization (PPO). Each of these methods help represent a distinct strategy for optimization and learning. DQN is a model that is a value-based reinforcement learning that uses a neural network to approximate Q-values and guide the agent's actions. Genetic algorithm is an adaptive heuristic model that uses a population of agents using mutation and crossover, which is guided by a fitness function that rewards only favorable gameplay. PPO is a policy gradient method that optimizes an agent's policy by interacting with the environment and updating action probabilities.

After running our 3 different agents of all our models for multiple days, our results demonstrated the strengths and limitations of all our different approaches. The genetic algorithm achieved the highest score of 10 million as it excelled in discovering unique and very effective strategies through its slight random exploration using large populations. The DQN managed to achieve a score of 4 million, as it traversed through neural networks. However this model could most definitely score better if better computational resources were provided instead of our own individual computers. PPO, on the other hand, struggled to generate any meaningful results as we struggled to stabilize policy gradients. Overall this project showed us the trade-offs between these techniques, with population based expiration and the potential of deep reinforcement learning methods for mastering a game like Tetris.

2. Related Work

DQN is the most researched reinforcement learning technique for Tetris. This is because it makes the most sense for a game where the action and state spaces are discrete. In Tetris, it is always possible to calculate the value of taking a specific action in a specific state. We followed the approach to game space representation from [1], which simplifies the action space by compressing actions to the final placement of tetrominoes, rather than the individual movements that led there. [1] also found that learning could be improved by switching from the default quadratic reward system, where clearing lines are rewarded by the function: r = 1 + (lines cleared ** 2), to a linear reward system. This is because with a quadratic reward, the agent is incentivised to take more risks to clear many lines once, leading to more frequent game overs. Additionally, we based our neural network off of [2], and experimented with action space representation using ideas from both [1] and [2].

Genetic algorithms have been applied to Tetris many times. This is due to the ability of easily being able to optimize gameplay strategies with population-based exploration. Unlike reinforcement learning, GAs does not require step by step learning, but can however evolve agents through a simple fitness function. We followed the approach described in [3], which emphasized rewarding rows cleared, penalizing gaps, and balancing column heights. This method requires the use of crossover and mutation to be able to create new agents that use new strategies, enabling the ability to discover many unconventional solutions. Inspired by [3] we tailored our GA to heuristics, tuning our mutation and crossover rates in order to achieve the highest performing agent possible, exceeding traditional RL methods.

3. Data Sets

We initially planned to use OpenAI's Tetris-Gymnasium for all of our algorithms. However, we struggled to interface with this environment due to overly complex code and large game state spaces. In the end, we only used this environment for our attempt at PPO. We switched to a Tetris environment for reinforcement learning created by Viet Nguyen [2]. This environment compresses actions into the final placement of the tetromino. Since each tetromino has 4 possible rotations and the board width is 10, this results in **40 possible actions per each state** (with some actions being illegal). This environment represents the state observation space simply as four numbers: the number of cleared rows, the number of holes, the board bumpiness, and the board height. We knew that we wanted a simplified state representation rather than processing the entire board, so already having this in the environment allowed us to spend more time on our algorithms. Additionally, it was easy to modify the state space to add additional features and to modify the reward system for experimentation in this environment. Figure 1 illustrates example states of the Tetris board during the gameplay, showing the cleared rows and gaps as key features, with total pieces used, score, and lines cleared as stats.

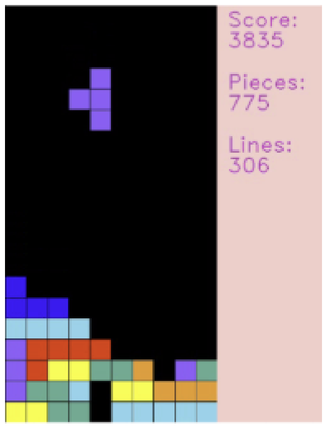

Figure 1. Tetris Environment

4. Description of Technical Approach

Deep Q-Networks (DQN):

Our DQN training setup was quite standard. We used an epsilon-greedy training loop with an experience replay buffer. The model was fit, after each game run, to batches sampled from the replay buffer, starting after the buffer stored 28000 experiences. A high batch size of 512 was used for sampling [1]. The Adam optimizer and mean squared error loss function were utilized for our training. DQN uses a neural network to calculate (estimate) the value of each state-action pair.

For implementation of our network, we created a simple multi-layer perceptron with three layers using ReLU activation. Xavier initialization was used to create stable weights and prevent vanishing/exploding gradient issues. One interesting detail about our network is that the output is one single value, the selected action, rather than the value for all of the possible actions. This is due to the deterministic nature of Tetris, where you know exactly what the next state will be, after taking an action in the previous state. We also created a similar, alternative neural network that had larger layers, for when we experimented with increasing the state representation from four float values to eight.

Genetic Algorithm (GA):

Our GA setup was designed to create an agent that would play Tetris as long as possible by evolving a population of agents over successive generations. Each generation began with a population of 16 agents. Each of these individual agents were evaluated by using a fitness function that used the same reward structure as the DQN agent. This fitness function was based on positive rewards for rows cleared and minimized gaps, with a negative reward for height differences of the board. After all the agents in generation had reached their end state. Each successive generation had the top 20% of agents (elite fraction) to be preserved, while each new agent was generated through crossover (70% rate) and mutation (10% rate). Crossover is a process in which the weights of two parent agents produce an offspring. Mutation introduced Gaussian noise to individual weights of some of the offspring.

The fitness function was crucial as it acted as the optimization target as it guided the selection of agents for reproduction after death. This function combined gameplay metrics to provide a single score for each individual agent. This score is what helped to determine the likelihood for the agent to be chosen for reproduction. In our case we trained across approximately 30 generations, with each generation using 16 agents. From these agents the top 3 agents were used as parents, with their children being used for the next generation.

Unlike gradient-based methods, GA used population-based search, which allowed for exploration of many unconventional strategies through mutation and crossover of the parent agents for each successive generation. However, this comes with a slight issue of this causing the results to be more variable across each run, making GA to be more random to training when compared to DQN.

Proximal Policy Optimization (PPO):

PPO is typically used for environments that have continuous state and action spaces, so it has not been tested for Tetris. However, we were curious to see if it could potentially perform well in high score scenarios where fast decisions are needed. Unfortunately, we were unable to get our PPO model to learn effectively. For our approach, we created a simple neural network with two heads, one for the value of the state-action pair (critic) and one for computing the probabilities of the possible actions (critic). Additionally, we implemented a clipped objective function to prevent the policy from being updated by more than 40% at once. Functions were also designed to calculate the discounted future rewards of states and the Generalized Advantage Estimate to determine the agent’s moves. PPO’s failure may have been due to errors in implementing these functions that we ran out of time to correct.

4. Software

Tetris Environment:

tetris.py:

- Adapted from user made Tetris environment [2]
- The original environment's reward system was changed to reward lines clear linearly rather than quadratically. The state space was also experimented with and made more complex.
- The environment was adapted for GA to be able to handle the population based approach.

Deep Q-Network (DQN):

All code in *dqn.ipynb*

Class/Function	Functionality
DQN	Torch neural network consisting of 3 sequential layers, utilizing ReLU activation and batch normalization. The output is the selected action index to select. Xavier initialization is applied to the weights. There are two implementations of DQN, one with size 64 layers for the environment with 4 features, and one with size 128 layers for the environment with 8 features.
train	Runs game simulations, choosing either the best known action or a random action, based on the epsilon-greedy algorithm. State-action pairs and their reward are stored in an experience replay before. After 30,000 experiences are stored, the model begins fitting to sampled experiences. Loop runs for 3,000 games (epochs) but can be terminated early.
record_game	Records a single game that plays using the trained model, utilizing opencv library
evaluate_model	Runs *num_games* games and returns the average score, average tetrominoes survived, and average lines cleared.

Genetic Algorithm (GA):

File Name	Functionality
agent_genetic.py	Implements the GeneticAgent class, which is used to represent the individual agents in the population. All these agents use a weight vector to decide the game states and the select actions.
genetic_algorithm.py	Contains the GeneticAlgorithm class, which contains the process of the evolution of the population of agents. This evolution uses a different implementation for crossover, mutation, fitness function, and fraction elitism to be able to improve the gameplay agent over multiple generations. Crossover (crossover(parent1, parent2) combines the weight of two parents to create an offspring. Mutation (mutate(agent)) uses Gaussian noise to the agent in order to allow for more random exploration.

run_genetic.py	This is the main training loop of the genetic algorithm by tracking the fitness trends of each successive generation, through plotting on a line graph. This also saves the best performing agent model with pickle.
run_best_agent.py	After a fitness agent has been trained, this loop loads and tests the best performing agent that was saved. This can visualize the game in real time, by saving the whole game as an mp4.

Proximal Policy Optimization (PPO):

Attempt to train PPO model onto tetris is located in *ppo.ipynb*. However this was a failed attempt due to lack of time and resources. However this is good starter code for us to be able to develop onto more in the future.

5. Experiments and Evaluation

Deep Q-Network (DQN):

For our DQN model, we experimented with changing our reward system from quadratic to linear and with increasing the state observation space from 4 values to 8 values. This resulted in the following 3 adaptations that we trained:

- Adaptation 1 (trained for 2k epochs)
  - State space: Lines cleared, Holes, Bumpiness, Height
  - Reward: Quadratic
  - Avg Score: 31, Avg Tetrominoes: 32, Avg Lines: 0.1
- Adaptation 2 (trained for 1850 epochs)
  - State space: Lines cleared, Holes, Bumpiness, Height
  - Reward: Linear
  - Avg Score: 488832, Avg Tetrominoes: 97793, Avg Lines: 39104
- Adaptation 3 (trained for 2 epochs)
  - State space: Lines cleared, Holes, Bumpiness, Height, Row Transitions, Column transitions, Cumulative Wells, Hole Depth
  - Reward: Linear
  - Avg Score: 446, Avg Tetrominoes: 118, Avg Lines: 33

Reasoning for Linear vs. Quadratic Scoring:

Quadratic scoring is based on the function *r = 1 + (lines_clear ** 2) * board_width*. This rewards the agent significantly more for clearing as many lines as possible at once. However, this also incentivizes the agent to play more risky, leaving large gaps, highly decreasing its rate of survival. This is demonstrated in figure 2, where the agent tried to form a large "well" but lost before it could clear the lines.

Once changed to a linear scoring function of *r = 1 + (lines_cleared) * board_width*, the agent cared much less about clearing a high number of lines and played more safe. This is why adaption 2 far out performed adaption 1 even with less training.

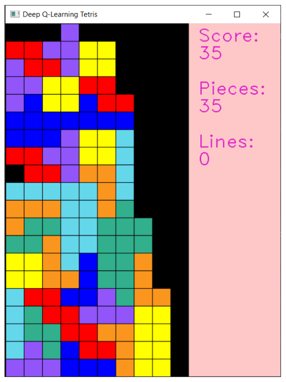

Figure 2. Quadratic failure

Reasoning for expanding observation space:

The environment [2] we used, initially represented the observation space as simply: rows cleared, number of holes, board bumpiness, and board height. We were worried that this might be too simplified to fully represent the Tetris board. Additionally, [1] used 8 features for their observation, so we added 4 of their features to our environment: **cumulative well depth, the number of row transitions, the number of column transitions, and cumulative hole depth**. However, our model that utilized these additional features (adaptation 3) performed worse than the one with just the base 4 (adaption 2). This could be because of poor implementation of these features or training limitations due to time.

Additional Notes:

Due to the fact that we had three different DQN adaptations and each game can take very long once the model is performing well, we were not able to train each model as long as we wanted to. We had wanted to train each one 3k epochs originally but were only able to do this adaptation 2 (see figure 3). Additionally, the highest observed score, 4293852, came from adaptation 1 at 2650 epochs. Because of this, it is possible that with long enough training, quadratic scoring could outperform linear scoring.

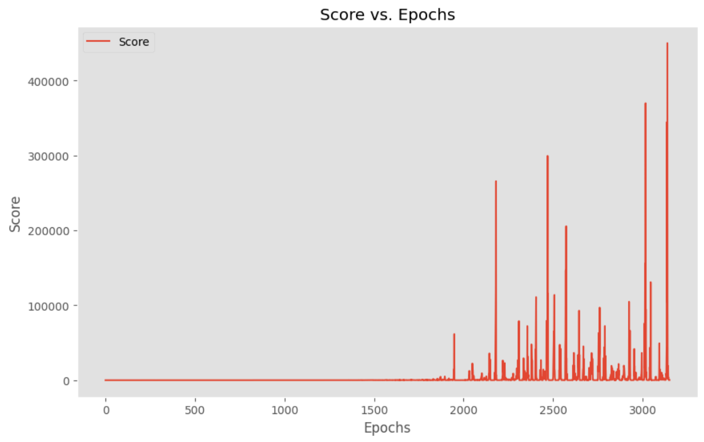

Figure 3: Adaptation 2 trained for 3k epochs

Genetic Algorithm (GA):

Similar to the DQN models, we evaluated the performance of GA through multiple different metrics. We focused on tuning the hyperparameters and fitness function to optimize the agent to play Tetris. This section we will discuss our experiments with hyperparameter tuning, fitness function design, and the results that we observed.

Hyperparameter Tuning:

We explored different set of numbers for the following parameters:

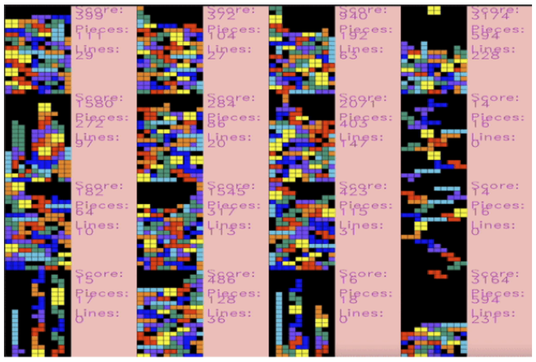
Figure 4. Genetic Population Render

- **Population Size:** This determines the number of agents competing in each generation. We tested population sizes ranging from 8 to 40. A larger population does an excellent job when it comes to exploration. However the larger the population the more computational resources are required. This caused us to settle down to using a population size of 16 for each generation, as that was the max number of agents that an M1 Mac could handle before python would crash [4].
- **Crossover Rate:** This controls the frequency in which the genetic information between each parent is exchanged to create offspring. We experimented with crossover rates between 50% to 70%. Higher rates encouraged more exploration of new strategies, while lower rates emphasize exploiting existing strategies of the parent agents. After multiple runs we determined that a crossover rate of 70% yielded the best results.
- **Mutation Rate:** This determines the probability of random variations in an agent's weight during the reproduction after each generation. This is crucial in GA as it helps balance exploration and exploitation. We experimented with mutation rates between 1% to 15%. In this we realized that higher mutation rates introduced more randomness, while lower rates helped maintain stability. However it is important that lower rates lead to a premature convergence in the score. We found that in our case with a population of 16, a mutation rate of 10% provided the best results.
- **Elite Fraction:** This parameter determines the proportion of the top performing agents to be used for the next generation from the total population. This is important to determine what genetic material should be preserved. We tested an elite fraction of between 10% to 30%. Through multiple tests it was determined that 20% was the best proportion to get the best results.

Fitness Function:

The fitness function acted as a very crucial part of our GA. This function was used to determine the performance of all agents. The function rewarded agents for clearing rows while penalizing gaps and uneven column heights. This helped to ensure that there would be balance between the much needed short-term rewards and long term survivability for a game like Tetris. After much experimentation with different environments we landed on a fitness function with the following metrics, which are very similar to the DQN implementation:

- **Rows Cleared:** Reward agent for clearing rows, with much higher weights given to agents who clear multiple rows at once.
- **Gaps in Board:** Penalizing agents for creating or leaving gaps in the board. Gaps prevent rows from being cleared until all the rows above it are cleared.
- **Bumpiness (Column Height):** Penalizing agents for uneven column heights or very high column heights. This is done to encourage the agent to maintain a smoother board during the game.

- **Game Duration:** This is an effect from all the previous rewards as the other metrics will introduce higher fitness scores thus more rows are cleared thus the game lasts longer.

Experimental Results:

For the final run that we did for this research project, we ran the agent for 27 generations with the most optimal hyperparameters that were tested as stated in the Hyperparameter Tuning section. The best agent achieved a score of 10 million with an average population fitness of 2.2 million in that generation. This best performing agent was saved into a pickle file, to be used to play any tetris game in the future. Figure 5 shows the fitness trends (maximum, average, and minimum scores) across all the generations. Notably, fitness improved rapidly in earlier generations, particularly in our case from generation 1 to 7, where there was an agent who managed to score close to 7 million. This rapid improvement displays the powerful nature of genetic algorithms of being capable to explore diverse strategies early in the process. However, there are many noticeable dips within the scores after generation 7, as the agents begin to explore and converge. These results show the random nature of genetic algorithms, as convergence and mutation has a massive impact on the performance of the best agent. Many times exploration sacrifices the short term performance for the potential to discover better long term solutions.

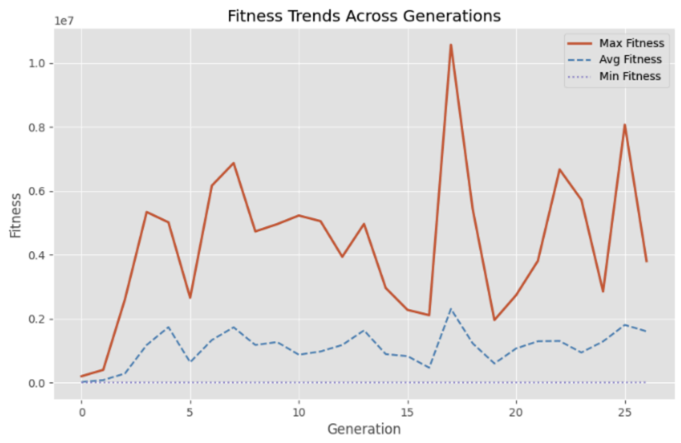

Figure 5. Generational Trends

6. Discussion and Conclusion

From this project we learned many new ideas and concepts about applying reinforcement learning to games. Our Genetic Algorithm (GA) and Deep Q Network (DQN) algorithms both performed exceptionally well. Each method demonstrated its own strengths and weaknesses. GA proved to be strong in exploration and adaptability, but had limited possible refinements and would over specialize. DQN showed strong generalization ability, but was computationally expensive and required a carefully crafted environment and reward system. We were unable to get our Proximal Policy Optimization (PPO) model to work, as we decided to focus on the other two. This is definitely something that we are interested in revisiting in the future. If done correctly, PPO could achieve unique and impressive results.

GA achieved our highest score of 10 million, yet we believe DQN is the best model because it is more stable and still achieved a score of 4 million. We are able to base this strong hypothesis based on the computational limits that we had faced when it came to training DQN. Since DQN is much more capable of consistent progress over time, we can determine that with more epochs, finetuning of hyperparameters, and time spent running the training algorithm, DQN should be able to outperform GA. Some unexpected findings are that our model performances would often drop off after peaking, especially for GA. This is what is called forgetfulness, as oftentimes the agents would forget effective strategies and become hyper fixated on certain patterns. We also learned the importance of choosing our reward system and how it can affect specific behaviors or patterns our agent will learn.

In the future, with more resources and time, we would love to explore the possibility of creating a hybrid model between GA and DQN. By combining the exploration capabilities of GA with the generalization and learning efficiency of DQN. We would be able to create a really strong and robust agent, possibly surpassing the approx. 10 million high score that we had achieved.

References:

1. Chen, "Playing Tetris with deep reinforcement learning" *University of Illinois Urbana-Champaign*, 2021, https://www.ideals.illinois.edu/items/118525 .
2. Viet Nguyen (nhviet1009@gmail.com), https://github.com/vietnh1009/Tetris-deep-Q-learning-pytorch
3. Duc Anh Bui, "Beating the World Record in Tetris GB with Genetic Algorithms." *Towards Data Science,* 2020, https://towardsdatascience.com/beating-the-world-record-in-Tetris-gb-with-genetics-algorithm-6c0b2f5ace9b

Appendix

A. Tools and Libraries Used

1. **Programming Language:** Python 3.9
2. **Libraries and Frameworks**:
    - **NumPy:** Efficient numerical computations for matrix operations and agent weights.
    - **PyTorch:** Neural network construction and training for DQN and PPO agents.
    - **Matplotlib:** Visualizing fitness trends and other metrics.
    - **OpenCV:** Rendering and recording Tetris gameplay for evaluation.
    - **Pickle:** Saving and loading trained models for reproducibility.
    - **Viet Nguyen's Tetris Environment:** Adapted for reinforcement learning and genetic algorithms.

B. Hyperparameters and Configuration

1. **DQN Parameters:**
    - Epsilon decay for exploration-exploitation tradeoff.
    - Batch size: 512 experiences per training step.
    - Reward systems: Quadratic and linear scoring.
2. **Genetic Algorithm Parameters:**
    - Population Size: 16
    - Mutation Rate: 10%
    - Crossover Rate: 70%
    - Elite Fraction: 20%
3. **PPO Parameters:**
    - Clipped Objective Function: Ensures policy stability.
    - Discounted Future Rewards: Guides agent learning.

C. Environment State Representation

1. **Observation Space:** Comprised of numerical features:
    - Rows Cleared
    - Number of Holes
    - Board Bumpiness
    - Board Height
2. **Action Space:** Simplified to 40 possible actions (rotations and placements).

D. Key Metrics

1. **Evaluation Metrics:**
    - **Score:** Total points achieved during gameplay.
    - **Rows Cleared:** Measures gameplay efficiency.
    - **Fitness:** Combines multiple gameplay aspects into a single evaluative metric.
    - **Game Duration:** Tracks how long an agent survives.
2. **Experimental Metrics:**

- **DQN:** Linear vs. Quadratic reward impact.
- **GA:** Effects of mutation and crossover on fitness trends.
- **PPO:** Training stability with clipped objective functions.

E. Figures and Visuals

1. **Fitness Trends:**
    - Graph showing maximum, average, and minimum fitness across generations.
2. **Tetris Gameplay Snapshots:**
    - Screenshots illustrating agent strategies (e.g., minimizing gaps, clearing multiple rows).

F. Repository and References

- **GitHub Repository:** Tetris AI Repository

Referenced Works:

- Chen, "Playing Tetris with deep reinforcement learning"
- Viet Nguyen, "Tetris-deep-Q-learning"
- Duc Anh Bui, "Beating the World Record in Tetris GB with Genetic Algorithms"