A NEAT Approach to Tetris

Alex Andrew Josh Dudley Tony Lau Miles Shamo

Name	NetID	Question asked to Group #	Question answered (name)
Alex Andrew	aandre26	17	N/A
Josh Dudley	jdudle4	21	N/A
Tony Lau	tlau7	7	Melville Misayah
Miles Shamo	miless2	2	Sarang Gawane

Abstract

We attempt to apply NeuroEvolution of Augmenting Topologies to a competitive Tetris environment with a variety of game state representations, fitness functions, and other hyperparameter configurations. Unfortunately, we were not able to create a successful model. To confirm our methods, we retried many of our experiments on more traditional, single player Tetris, encountering the same obstacles as in the two player game.

1 Introduction

1.1 Tetris

Tetris is a famous tile matching puzzle game that is played around the world. Since its original release, versions of Tetris have been developed for almost every gaming console and computer. Typically it is played on a rectangular 10x20 grid of squares. The player is responsible for rotating and shifting pieces called tetrominoes as they fall towards the bottom of the play area. Tetrominoes themselves are contiguous pieces made of four squares, each in one of seven different patterns (Figure 1). Players earn points by creating full horizontal lines across the play area. That line is then removed from the play area and all remaining pieces are moved down. While the player can clear single lines to increase their score, clearing multiple lines simultaneously will give the player a higher score than individual clears. Play continues until the play area no longer has the space available to bring in a new piece at the top of the field.

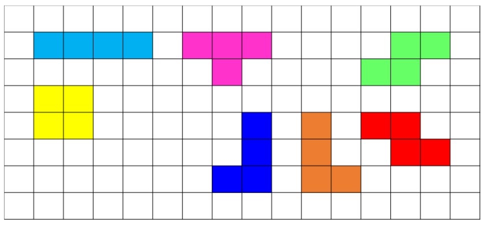

Figure 1: The seven tetrominoes of Tetris.

While the original version of Tetris spawns these tetrominos entirely randomly, modern versions have changed those methods to ensure the player would not be subject to floods and droughts of specific tetrominoes in order to make gameplay more consistent for a skilled player.

Likely due to its popularity, Tetris is a surprisingly well studied problem in machine learning with attempts at models dating back to the nineties[4]. Throughout, the most prevalent approach has been to train an evaluator which, given all possible board states that can be reached by placing the current piece on the board, rates each and takes the highest rated position as its next move. However, there have been countless other approaches including: hand crafted algorithms, cross-entropy, reinforncement learning, and modified policy iteration. Comparing between them can be challenging, as studies generally implement their own version of Tetris with various minor differences, but a good model will clear hundreds of thousands of lines on average.

1.2 Competitive Tetris

Tetris is typically played by a single player on a single field. With the advent of competitive Tetris came numerous changes with how the game is played. In some modern iterations the game can support up to 99 players simultaneously playing on their own fields[2]. All players are given the same pieces in the same order to ensure that there is a level playing field. Instead of score being the measure of a player's success, the goal of these versions is typically to survive longer than your opponents. As such, gaining a higher score is no longer the only reward for multiple lines being cleared simultaneously. Alongside score, horizontal lines, called "garbage," are sent to the bottom of the opponent's field when multiple lines are cleared in your own game, pushing the pieces in the opponent's field upward. These garbage lines are missing a single square and can be cleared quickly given the right pieces. The goal of this mechanic is to push one's opponent closer to the top of the screen and thus to defeat; the winner of competitive Tetris is the last person standing. Score is often still tracked, but is less relevant as a metric for performance than in traditional Tetris.

While score is still a useful indicator of an agent's skill, it is not the whole story in competitive Tetris. Clearing single lines traditionally sends no garbage to the opponent so strategies revolve around creating setups to quickly achieve multiple Tetris, T-Spins—rotating the T tetromino into specifically shaped holes in the board to clear lines—or clearing lines with consecutive pieces to form a combo to send a large amount of garbage to the opponent.

Unfortunately, executing these higher level strategies requires a very accurate model of the board state, which is not always compatible with simplified representations. For example, a common portion of the representation used is the "Column Height," the number of blocks from the bottom until the first block of a Tetromino is found[8]. This approach makes holes in each column (empty spaces) invisible to the agent. Such a simplification is usually harmless, but performing a T-Spin (Figure 2) requires the intentional creation of holes. An agent which cannot see holes would find it extremely difficult to intentionally execute a T-Spin. Tetris itself is well studied, but extending the problem to competitive Tetris is not well researched in the literature. It has been attempted, but usually in the form of "Classic Tetris" which is simply a high score competition between two players, devoid of the differences we have described prior[5]. We make a distinction between the two problems simply because a good Tetris player is a good opponent in a score contest, while a good Tetris player is not necessarily a good opponent under the modern, competitive Tetris ruleset.

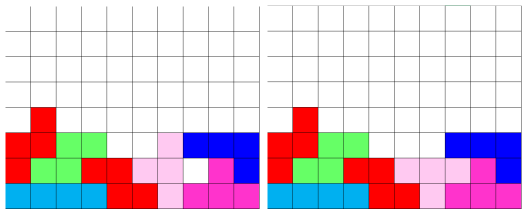
Figure 2: A T-Spin is where the T tetromino (here, colored pink) is rotated so that it fills an overhang in an adjacent column, such as the rotation shown from left to right, while completing at least one line. A column-height based representation would not give any indication such a hole existed.

1.3 Genetic Algorithms

When considering methods to train a model for Competitive Tetris specifically, genetic algorithms seem like a perfect fit. Not only are genetic algorithms extensively used in games and virtual environments[9][12], but they utilize a population of networks or actors which are all unique. As competitive Tetris requires an opponent and training the AI against its exact doppelganger would teach it little, we suspected having a large pool of actors would allow for better training. Games can be played between randomly paired actors and each can receive their own fitness score based on their performance in the game.

1.4 NeuroEvolution of Augmenting Topologies

For a particular genetic algorithm, we settled on using NeuroEvolution of Augmenting Topologies or NEAT[14]. This is perhaps the most well known genetic algorithm, we suspect due to the balance between power and conceptual complexity it provides, as opposed to its extension, HyperNEAT[13], or other alternative algorithms. Furthermore, due to its popularity, NEAT has been used to train standard Tetris before with success[15] and failure[11] so we had some prior work to compare our own, competitive Tetris training.

The NEAT algorithm was the first genetic algorithm to optimize a multi-layer perceptron by modifying both the weights and network topology. While prior algorithms modified the weights of the network to progress towards a solution, NEAT's primary innovation was a method of keeping track of each structural mutation in the network to allow properly crossing two networks—combining the characterstics of both in a mechanism modeled after sexual reproduction—with arbitrary topologies[14]. Thus, NEAT can start with simple perceptrons and allow mutations to slowly add hidden nodes and new connections. Beneficial complexity, structural mutations that could eventually improve the performance of the actor, are preserved via protections built into the algorithm, known as speciation[14]. This ensures that the networks generated by NEAT tend to search for new, more complex topologies over time, broadening the population's search space to more and more complex networks. As this improves performance, the overall population moves towards the more beneficial topology, continuing the search with that as a starting point.

Deciding what mutations and crossbreeds are beneficial is handled via a fitness function. The fitness function takes in an agent (usually as a representation that must be converted into the actual network) and returns a fitness value. The networks with the highest fitness values continue on to the next generation alongside new individuals created via crossbreeding members of the population with mutations. Networks with low fitness are removed from the population, with considerations in place to protect new structure as structural additions, adding nonlinearity, tend to initially lower performance until they are further tuned[14]. Each network has the possibility to mutate with the addition of hidden nodes, connections, and changes in weights. The result of these actions are three new groups which appear in the next generation, first an elitism group formed of high fitness individuals, a crossbred group, and a mutated group. NEAT differs from other genetic algorithms by having an initially minimal structure and adding complexity through the mutations over time.

2 Experimental Setup

For our experiment, we initially considered implementing the NEAT algorithm directly. However, while the actual approach of NEAT is fairly simple to comprehend, it is more difficult to implement efficiently than expected. As such, we used the NEAT-Python library[7] to focus on the problem itself rather than the implementation. The library allows us to specify hyperparameters via a configuration file and, importantly, supply our function that takes in the population and assigns each its fitness score.

To create such a function we, of course, need a game of Tetris. We implemented Tetris through two classes, the game board and piece. The game board contains the 20 by 10 board as a binary matrix, score, next piece and current piece. The Board class also manages enforcing the rules of Tetris; it detects collisions, illegal moves and when lines need to be broken. Piece contains the information about a piece, such as which Tetromino it is, it's current and next rotation, and its location on the board.

Once we set up our game of Tetris we then had to interlink the NEAT-Python with our Tetris implementation. This was done by specifying the input layer size to be the same as the vector of board representations, and having the output layer the same size as the 5 available moves, left, right, rotate, drop down one row, drop down until piece collides.

```python
def eval_single_genome(genome_id, genome, config):
    """
    runs 1 network to test it's fitness on XOR.
    :param genome_id: the genome's ID (unused here)
    :param genome: one genome from all the genomes
    :param config: the configuration for NEAT
    :return: the id and the fitness of the genome
    """
    genome.fitness = 4.0
    net = neat.nn.FeedForwardNetwork.create(genome, config)
    # test on each value of xor and subtract fitness for squared error
    for input, output in zip(xor_inputs, xor_outputs):
        answer = net.activate(input)
        genome.fitness -= (answer[0] - output[0]) ** 2
    return genome.fitness

def eval_pair_genomes(g1, g2, config):
    """
    evaluates 2 genomes, penalizing the worst by half its fitness
    """
    f1 = eval_single_genome(1, g1, config)
    f2 = eval_single_genome(2, g2, config)

    if f1 > f2:
        return f1, f2 / 2
    elif f2 > f1:
        return f1 / 2, f2
    else:
        return f1, f2
```

Figure 3: A punishing XOR fitness function. After evaluating each genome as normal, we halve the fitness score of the worse performing actor.

2.1 Modifications

Fortunately, NEAT-Python's implementation does not request a function of one agent to one fitness function. Instead, the library requests a function that takes in a list of all the agents in the population and assigns each one it's fitness score as an attribute. This is necessary as we would like to train the AI models in pairs. Thus, we were able to make random pairings of actors and play a game with each pairing. This allows us to train the AI actors directly in our competitive Tetris environment, instead of training them in isolation and only later testing them there.

As we expected we may need large populations in our experiments, so efficiency was a prime concern, especially as a single Tetris match can take many moves to resolve. To alleviate this issue, we parallelized the fitness function, splitting the games across multiple processes and merging the fitness results back to provide a fitness value for each genome. With this setup, we were able to pair up members of the population, let them compete in our competitive Tetris environment, and then provide a fitness function for each one, potentially even providing fitness based on the winner as well as performance.

Of course, with such a change to the algorithm, we had to verify that it was still functional. We did so by asking our system to learn XOR, in part because we could compare our results against the implementation's own XOR example provided with the library[1]. Importantly, to ensure that having interactions between random actors within the population didn't impact the effectiveness of the algorithm, we modified the XOR fitness function to penalize whichever actor performed worse in the pair. The function (Figure 3) halved the fitness of the network with lower performance. This is an extremely harsh punishment and provides no actual benefit to the training of the algorithm. This is the opposite of our intent in our actual experiments as we provided rewards for winning instead of punishments for losing. Nevertheless, the system is capable of training a model to compute XOR even with this punishment (Figure 4). As such, we moved forward, assuming that providing benefits to winners would be much less impactful than arbitrary punishments, especially as we would like to keep those actors that can consistently win over just those that can clear lines so rewarding winning makes conceptual sense.

3 Experiments

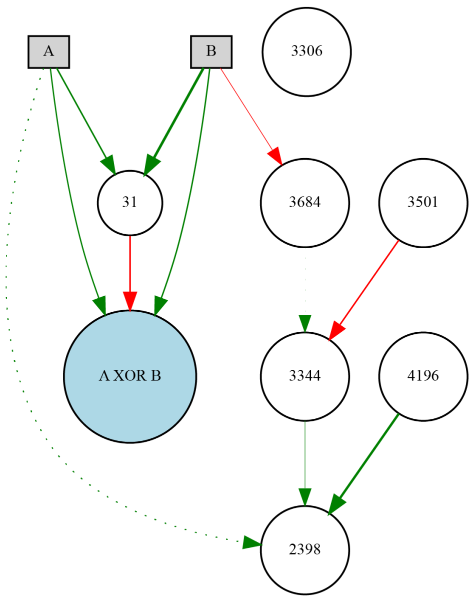

Figure 4: An example successful genome created with the above algorithm. Green lines are positive weights, red are negative, and their thickness is correlated to their magnitude. Dashed lines are disabled connections. On this particular run, we have a number of extranneous nodes that are not relevant to the working of the network.

Despite our tests with training models for XOR suggesting our approach was possible, we quickly found that our experiments failed to produce results. Of course, we did produce agents that are "capable" of playing Tetris but success stories could occasionally clear a single line or two. Ironically, this was a significant improvement from our earliest experiments, which would distribute the pieces more evenly in order to increase survival time and score, however none were able to consistently clear more than one line. We thus failed to produce an agent that is good at playing competitive Tetris.

Due to this failure, we analyzed similar projects and noticed that our approach had a feature that had been implemented previously, with results like our own. We had asked the network to supply the same inputs a human would (move left, move down, etc.) and found another project using NEAT in this fashion had similarly poor results[11]. Thus, we looked to see how we could modify the problem we asked our agents to solve, hoping to imitate approaches that did work. We reviewed another similar, but successful project[15] and attempted to mimic their framing of our problem. Instead of asking for human-like outputs from our network, we instead provided 40 outputs corresponding to a possible rotation of the piece at every column. After modifying our Tetris representation to be able to support this as in input, we attempted to train with this new method of interaction, placing pieces directly instead of asking for individual inputs. Unfortunately, we achieved no better results in any of our tests with this method.

We assumed, after our trials of both approaches with various representations, fitness functions, and hyperparameters, that this was likely a problem with the competitive Tetris setup specifically, despite it working well for XOR. Therefore, we tried the same experiments with a single player Tetris environment, modifying our competitive one to remove garbage and unlinking it from an opponent as well as removing as much complexity from our training setup (parallelization, pairwise evaluation, etc.) as possible. Unfortunately, our experiments had the same results; at best we had agents that would occasionally clear a line. While it was clear the agents had learned—some showed encouraging behaviors like partially avoiding large stacks of pieces to place pieces further down the board—we had no agents that could consistently clear lines. As such, what follows is documentation of the different parameters we tuned in our attempts. While these were tried in a number of different combinations, we list them here individually for clarity's sake.

3.1 Hyperparameters

Like any complex machine learning algorithm, NEAT has a wealth of hyperparameters that may be varied with besides the number of inputs, outputs, and its fitness function. The most important are the population size and the various mutation rates. As a larger population is often used for more challenging problems to help avoid a local minima[14], we attempted to increase the population size to help avoid being trapped in local minima. We initially started with a population size of 300 individuals and progressed to larger ones from there. However, up to population sizes of 1000 members, we saw no improvements. Similarly, we attempted to increase the evolution rates for new connections, new nodes, and weight alterations in the hopes of further driving the algorithm towards new strategies. Unfortunately, higher mutation rates also had little effect, even paired with a larger population to ensure mutations did not completely overwhelm any progress that had been made prior.

When analyzing the results from any run, we could generate a plot of the species of the network over time. Species are formed via the division of the population into groups of similar networks in order to protect new structural innovations and allow them the opportunity to lead to improvements[14]. It was common in runs to see many species be created, progress, then go extinct (Figure 5). As we suspected this pattern could be preventing improvement, we also attempted to tune the related hyperparameters. For NEAT-Python specifically, we can edit the allowable stagnation of a species, the number of generations a species is allowed to stagnate (no increase of its average fitness score) before it is removed from the population[3]. We tried raising this value from twenty-five generations to one hundred. This did tend to let species live longer but they continued to stagnate. After the initial few generations of each experiment, most new species would make no improvements until they were eventually wiped out by stagnation.

Perhaps, since we seemed to always end up trapped in a local minima, Tetris was simply too complex to learn with the processing time we had starting from a minimal perceptron. Off this hunch, we also experimented with adding hidden layers directly at the start of the algorithm. In these trials, our network would start as a 2 layer perceptron with a set hidden layer. This would then be used as the basis for NEAT's optimization. When experimenting with this, we configured the multilayer perceptron to only have links between adjacent layers at the start; the inputs linked to the hidden layer which linked to the outputs. Unfortunately, adding even one hundred hidden nodes did not change our results in any meaningful way.

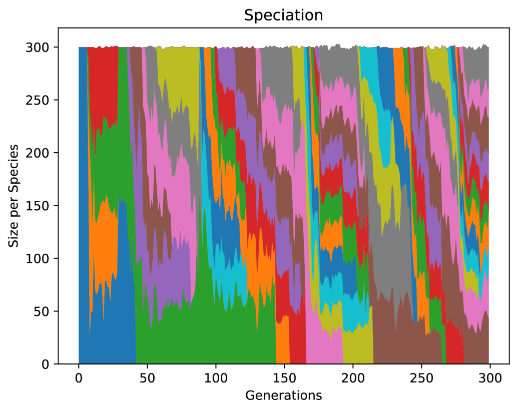

Figure 5: An example speciation graph from one of our trials. Each colored volume tracks the size of a species over successive generations. After the first 40 or so generations, we see a pattern of releatively consistent diagonal lines downwards. This implies that new species are being created regularly (they are placed at the top at their inception) and older species are consistently dying out. The consistent deaths here are largedly caused by stagnation.

3.2 Representations

In addition to NEAT's configuration, our experimentation required us to define the representation of the game state. A representation is a normalized, one dimensional vector containing information that describes the game state. For example, perhaps the simplest representation of Tetris would be to simply take the play grid as presented to the player and flatten it to a vector instead of a grid. Of course, the representations we experimented with were based on multiple individual features of the game state, instead of the entire abstract state.

The simplest to conceptualize but most complex representations are based off the raw game board. These representations take the flattened game board alongside other defining features, such as the current piece and the number of lines of garbage the player would receive (in the competitive case). While this style of representation is easy to conceptualize, it puts the burden of extracting useful information fully on NEAT's optimization.

Feature	Description
Raw	Flattened game board
Column Heights	The height of each column
Piece Shape	The heights and widths of the piece on a $4x4$ grid
Max Height	The maximumn Column Height
Holes	Number of squares that have filled squares above them
Bumpiness	Aggregate change in the height between adjacent columns: $bumpiness = \sum_{c=1}^{10} \lvert height_c - height_{c-1} \rvert$

Figure 6: The primary features we focused on in our experiments

To help guide the network, we created more concise representations of the game board by combining higher level features. These representations are attempts at simplifying the game state to provide more meaningful inputs to the individual networks, such as the height of the stack in each column, with the aim of reducing demand placed on the NEAT algorithm. To determine potential features to use, we considered features training other Tetris models[10][4]. We selected the a subset of these features (Figure 6) to focus our experimentation. Of note for competitive Tetris in particular, we also treated the number of lines of garbage currently being sent to the board as another feature that was included in every test. However, as sending a line of garbage requires clearing at least two lines at once, this feature was never of much use in our representations, seeing as they never consistently cleared multiple lines.

3.3 Fitness Functions

The fitness function in our experiment describes how well a game of Tetris was played; we experimented with a few ways in which a “good” game can be described. We first considered the simplest fitness function which was to use the ingame score counter. Using the counter as a fitness function resulted in short games where the network failed to stay alive long enough to clear lines. The next implementation we tried was score and ticks alive, which caused for a more even distribution of pieces and some accidentally cleared lines due to the distribution and longer game time. In an attempt to encourage the model to develop line clearing behavior we reduced the weight of game ticks, however this did not improve results, instead behavior appeared to be similar to just score.

A behavior we observed early on was what we will describe as thrashing, where the model rapidly sends inputs to a piece causing it to spasm left and right as well as rotate. In order to mitigate this behavior we tried to punish making an excessive quantity of moves within our fitness function since we theorized thrashing was caused by unnecessary complexity in the model. However, the model then chose to not make any moves at all, even when the punishment was made quite minor. Instead of using score, ticks alive, and the number of successful moves, we used only the score and pieces placed in an attempt to simplify the function. This resulted in similar results to score and ticks alive with the improvement that the model thrashed significantly less.

We also tried to encourage the model to clear lines by heavily weighting the number of lines cleared. The result was somewhat promising; some models cleared one line consistently. However, the board developed many holes and overhangs which made getting a second line a question of luck instead of intention.

In Other literature on single player Tetris, average points per move[10] or average lines cleared per piece[6] were examined, their optimization problems diverged from ours as their model rated game states already, so their focus was on efficiency rather than simply rating a game. Nonetheless we attempted their approaches to a fitness function, they did not produce good results, mostly because while their models were clearing lines consistently. Our models usually had 0 fitness due to not clearing any lines.

4 Conclusion

Despite our best efforts, we failed to produce an agent that was capable of playing competitive Tetris, or standard Tetris. Our best models learned to avoid stacking pieces too high, occasionally clearing lines in the process. While this is an improvement from our initial ones that simply stacked the first few pieces on one side and the rest on the other to prolong their existence, it is an underwhelming result. Regardless of fitness function, representation, parameters, or even the problem statement, our results did not improve beyond this local minimum.

Further efforts towards this goal will likely need to reevaluate our work. As there have been successful models trained using NEAT, perhaps further comparisons of our approaches with theirs will reveal either errors in implementation or requirements of our own design that differentiate our work from those that have succeeded. For an example of the latter, one can choose to reduce the number of available moves the model can perform in order to reduce the number of permutations considered in a 2 step look ahead[10]. In doing so they remove the ability to perform advanced setups that appear in competitive Tetris such as T-Spins.

If such models were developed for competitive Tetris, we may be able to answer the question that drove us to attempt this in the first place: can advanced Tetris strategies be learned by an AI model? As most Tetris models developed are focused solely on clearing lines while staying alive, such strategies are not encouraged. A model trained in a competitive environment as we attempted would be encouraged to learn these techniques, hopefully developing a more technical style of play. Current models are able to play Tetris beyond a human capability, clearing hundreds of thousands of lines, however they fail to implement higher level techniques found in competitive play. Whether or not a model can learn such advanced Tetris techniques, remains to be determined.

References

[1] Overview of the basic XOR example (xor2.py) — NEAT-python 0.92 documentation.

[2] Tetris® 99 for nintendo switch - nintendo game details.

[3] Welcome to NEAT-python's documentation! — NEAT-python 0.92 documentation.

[4] Simón Algorta and Özgür Şimşek. The game of tetris in machine learning.

[5] Max Bodoia and Arjun Puranik. Applying reinforcement learning to competitive tetris. page 5.

[6] Niko Bohm, Gabriella Kokai, and Stefan Mandl. An evolutionary approach to tetris. page 6.

[7] CodeReclaimers. CodeReclaimers/neat-python. original-date: 2015-09-26T22:59:53Z.

[8] Gijs Hendriks. The effect of state representation in reinforcement learning applied to tetris. page 9.

[9] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. 80(5):8091–8126.

[10] Jason Lewis. Playing tetris with genetic algorithms. page 4.

[11] MilanFIN. tetris-neat. original-date: 2021-02-28T14:16:38Z.

[12] Kenneth O. Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. 1(1):24–35.

[13] Kenneth O Stanley, David D'Ambrosio, and Jason Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. page 39.

[14] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. 10(2):99–127.

[15] Wenhan. NEAT-tetris. original-date: 2018-02-23T07:46:11Z.