Playing Tetris with Deep Reinforcement Learning

Matt Stevens
mslf@stanford.edu

Sabeek Pradhan
sabeekp@stanford.edu

Abstract

We used deep reinforcement learning to train an AI to play tetris using an approach similar to [7]. We use a convolutional neural network to estimate a Q function that describes the best action to take at each game state. This approach failed to converge when directly applied to predicting individual actions with no help from heuristics. However, we implemented many features that improved convergence. By grouping actions together, we allowed the Q network to focus on learning good game configurations instead of how to move pieces. By using transfer learning from a heuristic model we were able to greatly improve performance, having already learned relevant features about Tetris and optimal piece configurations. And by using prioritized sweeping, we were able to reduce the some of the inherent instability in learning to estimate single actions. Although we did not surpass explicitly featurized agents in terms of performance, we demonstrated the basic ability of a neural network to learn to play Tetris, and with greater training time we might have been able to beat even those benchmarks.

1. Introduction

The primary motivation for this project was a fun application of neural networks. Tetris is a well-known game that is loved and hated by many. Recent results from the team at DeepMind have shown that neural networks can have remarkable performance at game playing, using a minimal amount of prior information about the game. However, there has not yet been a successful attempt to use a convolutional neural network to learn to play Tetris. We hope to fill this gap.

Furthermore, research into solving control problems with neural networks and minimal explicit featurization may have applications outside of just video games, such as in self-driving cars or robotics.

Tetris is a game that involves placing pieces on top of each other so that they fill in a rectangular grid. A common human approach to this problem involves looking for open spaces that match the shape of the current piece in play. This is a ultimately a task in visual pattern detection and object recognition, so we believe that a convolutional neural network is a natural approach to solving this problem. Tetris also involves a fair degree of strategy, and recent advances in deep reinforcement learning have shown that convolutional neural networks can be trained to learn strategy.

2. Related Work

One of the seminal works in the field of deep reinforcement paper was DeepMind's 2013 paper, Playing Atari with Deep Reinforcement Learning [6]. This paper and their 2015 follow-up [7] served as our primary inspiration, both for the general idea applying computer vision to reinforcement learning for video games and for ideas on how to proceed with our model architecture. Their results are extremely impressive and state-of-the-art, with [7] achieving superhuman performance on over half the games tested and falling short primarily on games that require long-term strategic planning and have very sparse rewards, such as Montezuma's Revenge.

Several of the techniques that [6], [7], and our paper use have much longer histories. For example, experience replay, which stores the game-playing agent's experiences to be trained long after the original policy has been abandoned, was introduced in [4]. RMSProp, an optimization technique that naturally anneals learning rates down over time and focuses its learning rates on dimensions with more consistent and thus informative gradients, was first introduced in [9]. [6] and [7] recommended but did not implement prioritized sweeping as a way to select saved examples to train on; we implemented this in our paper. Prioritized sweeping, which trains the model on the examples with the largest error, was introduced in [8]; a more computationally efficient version, which we did not implement but would consider as an extension, was introduced in [1]. Finally, one optimization method we experimented with but that [6] and [7] did not mention was Adadelta. Adadelta, which is less hyperparameter-dependent than other optimization methods and anneals effective learning rates down based on how much a given dimension has already been updated, was introduced in [11].

Finally, it is worth noting that previous attempts to build

agents capable of playing tetris all use explicit featurization rather than proceeding from raw pixel data the way we do. Some examples are outlined in [2] and [5]. These previous approaches all use features relating to the contour and height of the game board and the number of holes present, so the heuristic function in [10] that we used to help train our network can be thought of as a simplified version building on the work in these papers.

3. Methods

3.1. Deep Reinforcement Learning Algorithm

For this project, we adopted a deep reinforcement approach very similar to the one used [6] and [7]. In this reinforcement learning approach, our neural network is used to approximate a function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi] \quad (1)$$

Where s_t is the state at time t, a_t is the action taken at time t, π is a policy function indicating what action to take given the current state and is followed at every step from $t+1$ onwards, r_t is the reward obtained by taking action a_t given state s_t, and γ is a discount factor. Thus, the Q function represents the highest expected discounted sum of future rewards achievable by following a fixed policy from states to actions. Since we know that the terms $\gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$ represent γ times the highest expected discounted sum of future rewards from the state at time $t+1$ and that this value does not depend on s_t, a_t, or r_t but simply on the state at time $t+1$, the Q function obeys the Bellman equation and can be simplified as

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right] \quad (2)$$

Where s' is the state (or possible states, if the process is non-deterministic) that results from taking action a given state s, and a' is the action taken given state s'.

3.1.1 Formalizing States and Actions

The data used for our learning algorithms consists of tuples of (state, action, reward, next state). The representation of the state, action, and reward has a large effect on convergence, and since our algorithms are agnostic to the details of these values (except for the input and output dimensions of our Q network), we examine multiple different representations of states, actions, and rewards. The state is an image of the current game screen. Actions can be either single actions - a single movement of a piece like moving one space to the left - or grouped actions - the entire sequence of actions a single piece takes as it drops to the bottom. For rewards, we implemented two scoring functions. One was the standard tetris scoring rules built into the MaTris library. The other was a heuristic function taken from another tetris AI project [10]. The reward was the change in score from one frame to the next.

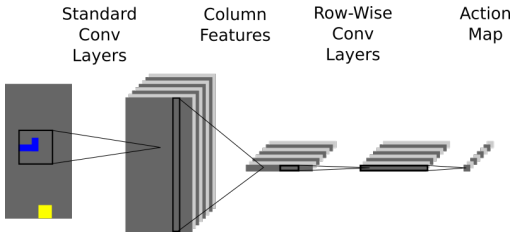

Figure 1. Schematic of our network architecture based on column features.

3.2. Network Architecture

Initially, we used a network architecture very similar to the one in [6] and [7]. The layers are, in order, conv5-32, conv3-64, 2 x 2 max pool, conv3-64, 2 x 2 max pool, FC-256, FC-256, and FC-13, with a ReLU non-linearity. However, we eventually switched to a new architecture that incorporated some domain-specific knowledge of tetris, namely that most relevant information is contained on a column-by-column basis. It is a well-known result in image processing that if a convolution is separable, then separating it into its horizontal and vertical components reduces both the computation required and the number of parameters. Reducing computation is clearly desirable, and reducing the number of parameters is desirable to avoid overfitting. This configuration is shown in Figure 1. Our new network architecture starts with conv3-32, conv3-32, and conv3-64 layers, followed by a layer that collapses each column into a single pixel with 64 feature channels. We then run conv3-128, conv1-128, and conv3-128 layers before feeding the output through FC-128, FC-512, and FC-13 or FC-1 (depending on the dimensionality of the final output). As before, we use a ReLU non-linearity. This network uses batchnorm and dropout with retention probability 0.75. We found that this architecture achieved similar results to our initial architecture but trained significantly faster.

3.3. Training

Since the Q function obeys the Bellman equation and has the form of Equation 2, we trained our neural network to match:

$$Q(s,a) = r + \gamma \max_{a'} Q'(s',a') \quad (3)$$

Where Q' is a cached version of the neural network. This cached version updates much more infrequently (once per

1000 iterations instead of every iteration) to prevent instability when training; otherwise, the network would be training on itself, and small initial disturbances could lead to drastically diverging q values.

3.3.1 Epsilon-greedy policy

For the bulk of our training, we used a standard epsilon-greedy policy, in which the tetris agent takes the estimated optimal action most of the time and a random action with probability ϵ. This ensures that the agent explore the search space and see how actions not currently considered optimal would have fared instead. Like [6] and [7], we anneal the value of ϵ from 1 to 0.1 over 1 million game frames.

In addition, we used a new approach when determining what action to take. A random tetris agent rarely clears any lines, so it will generate very few rewards. We implemented a separate, explicitly featurized tetris agent that was already trained to play well in order to increase the number of rewards generated in the training data. This agent selected whichever piece placement yielded the highest fitness function according to [10] (and if necessary, determined the moves required to achieve this). We then selected a random action with probability ϵ_{rand}, the action suggested by this second agent with probability ϵ_{agent}, and the action suggested by the neural network otherwise. This approach was heavily used in experiments where we used the change in game score as our reward function, as it would be highly difficult to attain enough rewards to train on just by following a standard epsilon-greedy policy.

3.3.2 Experience Replay

Like [6] and [7], we used a technique called experience replay to create a "replay memory" for our neural network to train on. While having the network play games, we saved every "experience", consisting of a (state, action, reward, next state) tuple, that the network had encountered, and we sample from this full breadth of experience when training the network, not just from the most recent game or frames. This complements the epsilon-greedy algorithm in helping the network continue to search the state space and avoid getting trapped in local optima; since the policy evolves over time, earlier frames will have a different distribution of actions for a given state than will later frames, so the network will still be able to train on off-policy results. Since the network is training on its estimate of the value of the next state, experiences from old policies can be used for training.

3.3.3 Prioritized Sweeping

Finally, we implemented a modified version of prioritized sweeping for sampling experiences from our replay memory. For every single experience consisting of s, a, r, s', we calculated

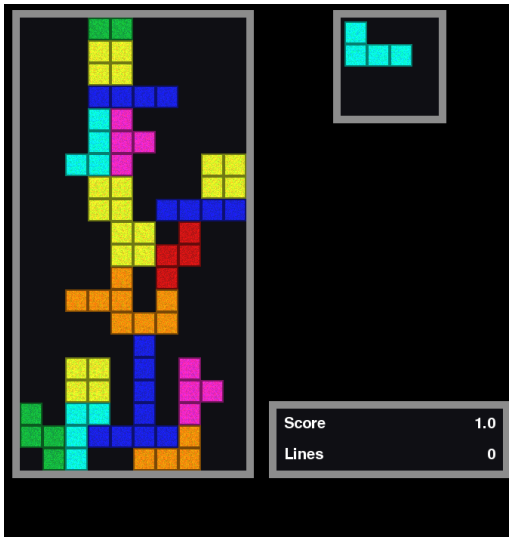

Figure 2. Sample of the emulator's output.

$$Priority = \left| Q(s,a) - r - \max_{a'} Q'(s',a') \right| \quad (4)$$

We then sampled experiences with probabilities proportional to their priorities. Thus, we trained the network on the experiences for which it had the greatest error and the most room to improve. We recalculated the priorities every time we updated the cached Q' network.

4. Dataset and Features

We use an emulator heavily based off of the MaTris library [3], a library for playing tetris written in pygame. This emulator was then modified to allow our algorithms to extract images and perform actions, and adjust the reward function. An example of the emulators output can be seen in Figure 2. This emulator was used to produce states, actions, and rewards for both training and validation. These states, actions, and rewards took on multiple different forms.

4.1. States

For the purposes of our project, the state of the game at time t is simply the pixel array of the game screen at that particular time. We experimented with using only a cropped version of the game screen that contained the actual game board but not the information on the side (including which piece was coming up next) and found this to not significantly impact either accuracy or training results.

In order to avoid unnecessary computation and memory usage, we downsample this image so that each block in the tetris game corresponds to one pixel. By reducing the size

of each state, this allows us to fit more training examples in our replay history without running out of memory, which is a crucial component of the experience replay technique we used. In addition, with lower-dimensional input, our network requires fewer matrix multiplications and fewer parameters in the fully connected layers, greatly improving speed.

4.2. Actions

We investigated two different ways of formalizing actions. One method, which was more similar to [6] and [7], was to map actions to raw button presses. In our case, we allowed 13 actions: a combination of one of four different rotational orientations and one of three different translational movements (left one column, right one column, or unchanged), plus the option to "hard drop" the piece and instantly send it to the bottom. In this case, the output of the neural network was a 13-valued vector where i'th element of the vector was the expected discounted reward of taking the i'th action given the input state.

Another method of formalizing actions we attempted was to have an action represent the final column placement and rotation of the piece. We call these grouped actions. We attempted this after running into some difficulties with formalizing actions as raw button presses—in particular, almost any individual move was perfectly reversible, and it took a lot of moves to get a reward, both of which made it more challenging to determine which moves really contributed to the final reward. Thus, this alternate method of formalizing actions more directly let us get to the heart of what we wanted the Tetris bot to really learn. In this case, since the number of actions was not constant, we could not have the neural network output a single vector containing the expected discounted rewards for each action. Instead, we rendered the image the game would display for each possible piece placement and fed each of those images into the neural network; the neural network then returned its estimate of the best expected discounted rewards attainable from that state, which we used to calculate the value of the action leading to that new state.

4.3. Rewards

Rewards are primarily based off of the tetris game score. Every time the agent clears lines, the reward is the number of lines cleared, squared. This means that a tetris is worth 16 points and a single line is worth one point. It was also found that including a gameover penalty improved the convergence of our algorithms, so we included this penalty in the game score. We also found that these rewards did not punish reversing moves, leading to pieces that vibrated and spun wildly on their way down. To deal with the third issue, we added a very small penalty (equal to between a tenth and a half of the average expected discounted reward) for taking an action that moved the piece. The reward after each move was the change in this score.

In order to give the network a good initialization point, we also implemented an intermediate scoring function that varies more smoothly between different moves than the game score and gives a meaningful reward after nearly every action. This initialization point was used for transfer learning by initializing the network on the smooth scoring function and then switching over to the game score function. This reward function uses an explicitly featurized fitness function from [10], which takes into account the aggregate height of the tetris grid (i.e. the sum of the heights of every column), the number of complete lines, the number of holes in the grid, and the "bumpiness" of the grid (i.e. the sum of the absolute differences in height between adjacent columns). The actual formula for this fitness function is

$$\begin{aligned} &-0.51 \times Height + 0.76 \times Lines \\ &-0.36 \times Holes - 0.18 \times Bumpiness \end{aligned} \tag{5}$$

And the reward was simply the change in this fitness function.

5. Results

5.1. Evaluation Metrics

Since Q learning is recursively defined, we did not consider fitting errors such as RMSE to be good metrics for evaluation. Instead, given our ultimate our goal to make a good game AI, we evaluated gameplay directly in order to gauge performance. We evaluated the performance of our algorithms using two metrics: game length and game score. Game length is the number of moves the player makes before they reach a gameover, and game score is the score based on the number of lines they clear. While the game score is ultimately more important, it can only be used to judge the performance of sufficiently advanced networks because there is a large initial barrier to overcome before the algorithm can score with any degree of consistency. Thus, game length, which was typically correlated with game score, is a good metric for showing intermediate performance since an algorithm can incrementally learn to survive longer.

In order to make our results more interpretable, we compare game performance to both a random agent and our gold standard, the agent from [10]. A random agent has an average score of zero, clearing a line less than once every thousand games. By contrast, the gold standard agent has an average score of about 200. In terms of game length, a random agent is easily measurable, so we report relative game length as a ratio of the game length of our agent to the game length of a corresponding random agent using a similar move structure. A random agent using grouped actions

Initialization	Actions	Rewards	Sampling	Average Score	Relative Game Length
Gold Standard[10]	-	-	-	200	21.7
Random	Grouped	Heuristic	Random	18.0	4.26
Transfer	Grouped	Game	Random	2.1	1.85
Random	Grouped	Game	Random	0.1	1.26
Random	Single	Game	Prioritized Sweeping	0	1.05
Random	Single	Heuristic	Random	0.01	0.79

Table 1. Experimental Results.

has an average game length of 230, and a random agent performing single actions has an average game length of 110. The difference is due to the fact that a random walk will tend to concentrate pieces in the middle of the game board, whereas grouped actions sample evenly across all positions on the game board and thus avoids building large towers of pieces too soon.

5.2. Experimental Setup

Our experiments lasted too long to do a rigorous hyperparameter search, so we performed a manual search instead. We used RMSProp with a squared gradient momentum of 0.95 to optimize our neural networks, with a learning rate of 2×10^{-6} that was manually annealed down to 5×10^{-7} when training appeared to stall. Although we experimented with it, we ultimately did not use any explicit L2 regularization, instead relying on batch normalization and dropout (with a retention probability of 0.75) for our regularization. The discount rates we used for Q Learning varied from 0.75 to 0.9 depending on the experiment. These values were selected to be high enough that rewards from clearing lines would still provide credit to the actions that preceded it yet low enough that the rewards of extremely distant line clearances would not affect moves too far back to have had a significant impact.

Our experiments were not run for a consistent amount of time. Experiments were terminated when they stopped improving, rather than at a fixed time point. Although this was not ideal, we had limited resources and did not want to waste them on models that were failing.

5.3. Experiments

For our experiments we sought to analyze three of our techniques for improving results: grouping actions, transfer learning, and prioritized sweeping. The results of our experiments are summarized in Table 1. Our results show that grouping actions makes the learning problem much more tractable. Grouping actions improved our relative game length from 5% better than random to 26% better than random when learning the Q function from scratch. It also showed better overall performance. Transfer learning also showed promising results, scoring an average of two lines per game, compared to effectively no lines for learning from scratch. Although our best result came from the network trained on our heuristic score function, our next best results came from using transfer learning and training our Q function directly from game scores. Although prioritized sweeping did comparatively the worst, even it showed signs of improvement, preventing the network both from acting totally randomly and from always defaulting to a single move like only dropping the piece.

A video of gameplay from our network trained on a heuristic scoring function can be viewed at `http://youtu.be/uzXMFUtvAlY`, and a video of our network trained via transfer learning can be viewed at `http://youtu.be/753mOxyES3I`. Both show fairly natural gameplay, but both also make several mistakes that would be obviously wrong to any human player of tetris. This could indicate that the network is overfitting, since it is sometimes unable to generalize from one kind of good move to another, or that the network is not yet sufficiently trained, since it is unable to recognize good moves sufficiently often.

5.4. Discussion

5.4.1 Grouping Actions

Grouping actions showed a significant increase in performance over trying to estimate single actions alone. This is not surprising, since it is a fundamentally easier problem. The single action problem involves learning both where to place a piece and how to get it to that position, while the grouped action problem involves only learning where to place the piece. Furthermore, when estimating actions, the reward must propagate through the chain of actions based on the Q value of the next state. The greater the length of this chain of actions, the greater degree of numerical precision is required, which means that achieving convergence will be much more difficult.

5.4.2 Transfer Learning

Transfer learning significantly improved the performance of our network. When switching from the heuristic scoring function to the actual one, the network had to readjust, and initially showed a large drop in performance, as

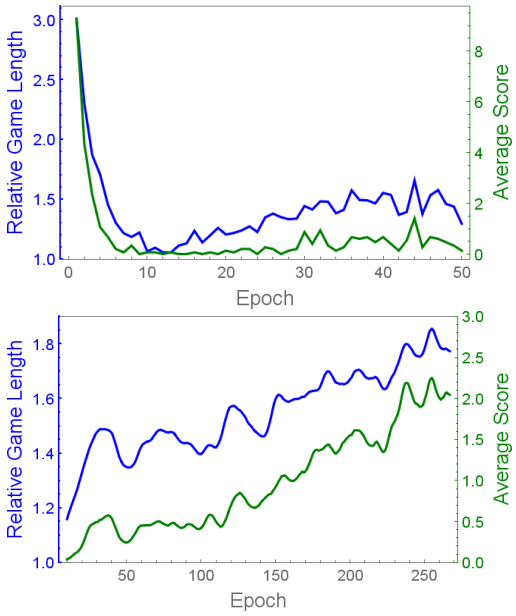

Figure 3. Performance of the transfer learning configuration over time. Top: Burn-in period for transfer learning when performance rapidly decays. Bottom: Gradual progress of transfer learning. Data are smoothed to better show trends.

is shown in the top panel of Figure 3. However, in spite of this huge drop in performance, the network started to score lines and improve its performance more quickly than the network trained from scratch on the game score. The performance of the network continued to improve over the course of training, as shown in the bottom panel of Figure 3. The reason for this improved performance is that the network likely learned many relevant features for tetris during its initial training, and after switching to the game score it only had to reweight these different features. Our results indicate that transfer learning is a viable strategy for training an AI for tetris. Although our agent did not surpass the agent that it was originally based off of, this may be possible with better tuning and longer training.

We were unable to investigate transfer learning for single actions because the network using the heuristic reward function failed to converge. The reward was the score of the piece if it were to be dropped at the current position, and as such the rewards were not smooth and strongly encouraged local optima. This is further evidence of the difficulty of training a tetris AI on single actions.

5.4.3 Prioritized Sweeping

When trying to train the network on single actions, we ran into several issues. One was the sparseness of rewards, and especially the difficulty of getting any rewards since random action is highly unlikely to lead to cleared rows.

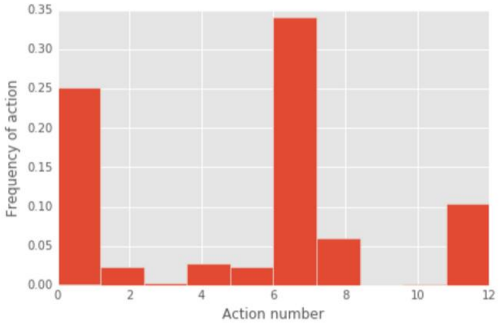

Figure 4. Prioritized sweeping smoothes the action histogram of the network.

While we were able to mitigate this with the our modified epsilon-greedy policy, we then faced the issue that certain actions (especially any rotations) were drastically under-represented in the experience dataset, meaning our network could not generate reasonable predictions for the value of these actions. We initially tried using Adadelta as our optimizer so that these under-represented actions would have larger effective learning rates and thus still converge. This worked well in small-scale experiments but failed when we tried to scale it up beyond small testing datasets.

Prioritized sweeping helped ameliorate these problems; as certain actions were under-represented in the sampling, their errors increased relative to other actions, giving them higher priorities and leading to them being more aggressively sampled. This helped ensure that the q-values for all actions were reasonably accurate and that no set of actions saw their q-values remain systematically low. Although action 0 (doing nothing) was roughly 83% of the actions in the original experience dataset, the distribution of actions recommended by the network was far more equitable, as seen in Figure 4. This indicates that prioritized sampling did its job and helped prevent initially under-represented actions from remaining under-represented.

Unfortunately, the actual game performance of the network trained with prioritized sweeping remained mediocre, as it failed to clear lines and was well within a standard deviation of the game length to be expected from playing randomly. However, we have reason to believe that this was simply an issue of training time. Recalculating the priorities after every epoch was especially computationally intensive, so epochs took far longer for prioritized sweeping than for the other two experiments. Thus, despite running the prioritized sweeping experiment for over 10 hours, we were only able to run about 15 epochs. The other training methods all took at least 50 epochs to attain decent performances and generally also had poor performance after only 15 epochs. Thus, if we had the time and resources to run the prioritized sweeping experiment for as many epochs as the other

two, we believe that prioritized sweeping could also have attained a more respectable performance.

It is worth noting, however, that the choices of hyperparameters can greatly impact the way the network learns to play. In particular, setting the penalty for moving too high can lead the network to learn to avoid ever making a move, leading to the network always recommending to do nothing and let the pieces stack up in the middle. Likewise, seeding the networks replay memory using a policy that hard drops a piece the instant it is in the optimal position leads the network to conclude that hard dropping pieces reduces the discount factor that is applied to future, always positive rewards, so the network hard drops pieces almost instantly and loses games in significantly less time than even playing randomly would.

6. Conclusion

Our results show that the key factor in the difficulty of a reinforcement learning problem is the time lag between the action and the reward. When training with grouped actions off of the heuristic reward function, the reward for a given action was immediate, and this network showed the best performance. The next best performance came from a network based off of grouped actions, where actions were only a few steps removed from the next reward. Our worst performance came from the networks trained to estimate actions, where actions were several dozen steps removed from the next rewards.

Intuitively speaking, a longer gap between action and reward means that a game involves a lot of strategy, and a shorter gap between action and reward means that a game is more reflex-based. We found that learning converges much more quickly for reflex-based games than for games that involve a large amount of strategy, which makes sense and is consistent with the pattern found in [7], where the DQN network performed much better on games like Pinball or Pong that have nearly instant rewards than on games like Montezumas Revenge with rewards that only manifest after long delays. Our results show that using an explicit AI to bridge the gap between strategy and reflex is a useful approach for improving gameplay performance.

6.1. Future Work

We believe that our results would have continued to improve given further training, although we had neither the time nor the resources to accomplish this. With further modifications, we could achieve convergence on a heuristic reward function for single actions, enabling us to investigate transfer learning for single actions. Modifications to the prioritized sweeping algorithm could potentially reduce training time and thus allow more epochs and improved performance. Using both of these approaches, we may be able to achieve performance significantly above random for estimating single actions, where we would have an end-to-end tetris AI based entirely on a convolutional neural network.

References

[1] D. Andre, N. Friedman, and R. Parr. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*. Citeseer, 1998.

[2] D. Carr. Applying reinforcement learning to tetris. *Department of Computer Science Rhodes University*, 2005.

[3] GitHub. Matris - a clone of tetris made using pygame. `https://github.com/SmartViking/MaTris`.

[4] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[5] N. Lundgaard and B. McKee. Reinforcement learning and neural networks for tetris. Technical report, Technical Report, University of Oklahoma, 2006.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, 1993.

[9] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:2, 2012.

[10] L. Yiyuan. Tetris ai - the (near) perfect bot. `https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`.

[11] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.