Tetris RL

Mohammad Khan (msk295)

Pablo Raigoza (pr428)

Shao Stassen (ses439)

`github.com/PabloRaigoza/TetrisRL`

1 Introduction to Tetris

Tetris is enjoyed by a wide range of audiences and has long served as a challenging benchmark for AI and machine learning research. Its dynamic state space, complex piece placements, and sparse reward signals make it an appealing but difficult environment for Reinforcement Learning. One way in which the reward structure for Tetris can be shaped is by giving rewards only when the agent clears a line. However, this could involve setting up complex board configurations with no immediate payoff. This sparse reward structure often hinders learning and requires techniques beyond just "throwing ML at it."

The challenge emerges because early policies frequently fail to discover even a moderately rewarding sequence of action before terminating. Basic policy gradient methods such as REINFORCE struggle to escape the initial low-reward plateau or decline back to low-reward after a lucky first go. Addressing this limitation is a key research question: can more informed initialization strategies lead to improved exploration and ultimately, higher quality policies?

With some initial research into the area, and lessons we learn from our own mistakes, we can conclude that training a Tetris RL agent often relies on handcrafted features, heuristics, or imitation learning from expert demonstration (such as BC and DAgger). Furthermore, human expert trajectories may not be optimal. A policy trained solely through BC and DAgger might inherit suboptimal biases from the expert. Our project aims to use the above lessons (and many more) to push the boundary of the Tetris agent space, by proposing and testing initializations, comparing their performance, and seeking to discover hidden information needed for a better Tetris agent.

2 OpenAI Environment Setup

We use the OpenAI gym environment for Tetris. Among the gyms in the final project page, it is one of the most popular and provides a (mostly) consistent and standardized API for RL. It also has a straightforward idea of defining the state and action space. Its default behavior is to have agents pick moves such as left, right, hard drop, and spin based on the current state of the board. It also provides the option for agents to see future pieces and swap current pieces with pieces held in a holder.

Our overall goal is to compare different initialization techniques such as random initialization, BC initialization, DAgger initialization, and a REINFORCE based initialization. In this case, random initialization means that the policy parameters are random with no prior domain knowledge. BC and DAgger initializations are just the weights of previously trained models, and a REINFORCE based initialization would be a modified policy gradient approach, incorporating an exploration incentive to deviate from previous policies, with the goal of discovering more diverse strategies before converging to a final policy. We thoroughly tested the first three of these initializations, while the last one had to be modified before it showed its effectiveness. Our idea is that initialization with a random policy will be the worst performing followed by initialization with BC, and finally initialization with DAgger. We hope that our custom initialization strategy will edge out all 3 of the above strategies.

Pablo and Shao collected ~ 90 expert trajectories over a week, with each trajectory having a length of 1000. This number was arbitrarily chosen because of the following observation. In a challenging environment like Tetris, a randomly initialized agent usually survives no more than 10-20 moves before achieving the "game-over" state. We believe that if an agent can survive 1000 moves, it is a sign that it is capable of consistently avoiding immediate game-over states, which demonstrates a baseline level of skill. Once an agent has achieved this baseline, we can utilize REINFORCE and other techniques to facilitate further improvement. Furthermore, if we can get agents that achieve 1000 moves played or more, these human baselines can provide more benchmarks such as "average reward per move." The rewards mapping we used was 1 per piece placed, $n^2 \cdot 10$ for n lines cleared in a move, -100 for game-over, and -10 for illegal moves. This was mostly given from the environment, but we increased the magnitude for illegal moves to further dis-incentivize them.

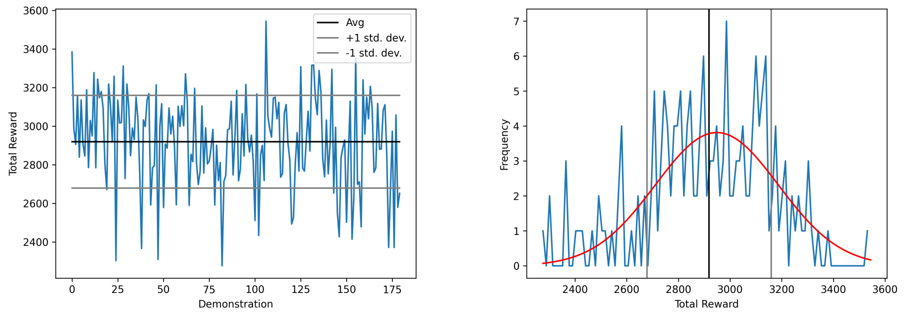


Figure 1: Distribution of Total Reward from Human Demonstrations

3 Initial Behavior Cloning & REINFORCE

The first agent model we used was a 3-layer (2 hidden layers) Neural Network (known as Mark 1/M1) with a dropout layer of $p = 0.2$ to preemptively counteract overfitting. This model took the mask of the active tetromino piece, the entire state of the board, the holder position, and pieces in the queue as 2D pixel arrays, and converted them into a 280 dimensional input vector. The output of the model, which is also the action space, was an 8 dimensional vector, with each coordinate corresponding to the logarithmic probability (logit) of a move like "move left," "rotate clockwise," "hard-drop," etc.

In our first attempt, we took the argmax of these probabilities to decide our action, but later we realized that sampling the distribution led to more stabilized training. We also tried both Cross Entropy (CE) and Binary Cross Entropy (BCE) as our loss functions, but ended up choosing CE because for most states (in our expert demonstrations) only one move is optimal instead of having many optimal moves. After training Behavior Cloning (BC) on the expert demonstrations, we saw the inklings of over fitting, however validation accuracy had yet to decline significantly. So we decided to increase model complexity (additional layers, promotes overfitting) and dropout to $p = 0.5$ (combats overfitting). We hoped that this aggressive change would aggressively increase validation accuracy because we noticed that even a single mistake could compound to larger errors.

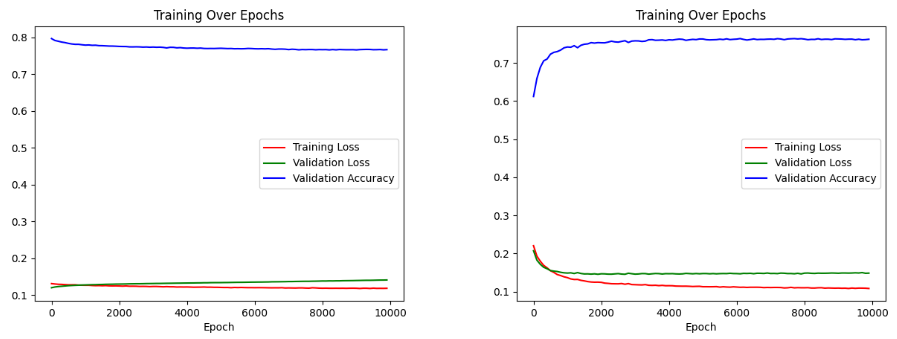


Figure 2: BC with Models M1 and M2 respectively.

We decided to wait until BC was more fleshed out before implementing DAgger as we wanted to test both DAgger with random initializations and BC initializations. At this stage, we also tested the M1 and M2 architectures with REINFORCE. However, REINFORCE was unable to surpass an initial reward threshold of -90 (-100 for dying and $+1$ for successful piece placement). Observing the game play, we realized that the agent stacked pieces directly on top of each other, leading to game-over in approximately 10 moves. This is because our REINFORCE models was unable to see the reward potential of clearing a line, instead it saw the quick reward of 1 from immediately "hard-dropping" a piece. If we were able to provide a strong initialization model, REINFORCE could have potentially learned of the high rewards associated with clearing a line; this strengthens our hypothesis that initialization models are important.

4 Expert Demonstrations for BC & DAgger

Our initial models, M1 and M2, struggled to achieve reliable performance even after thousands of epochs of BC training. Despite attaining roughly $\sim 75\%$ accuracy on the demonstration data, these models survived only around ~ 20 piece placements before failing. The core challenge was that executing a single optimal move in Tetris often requires correctly predicting an entire sequence of 5–10 actions. This meant that just one error in the trajectory resulted in wildly different rewards, an example of compounding errors. One solution to this problem is to drive validation accuracy to $\sim 99\%$ through more data. We realized that the amount of data we wanted might take weeks to obtain from human trajectories, so we decided to shift to obtain trajectories from an expert agent instead.

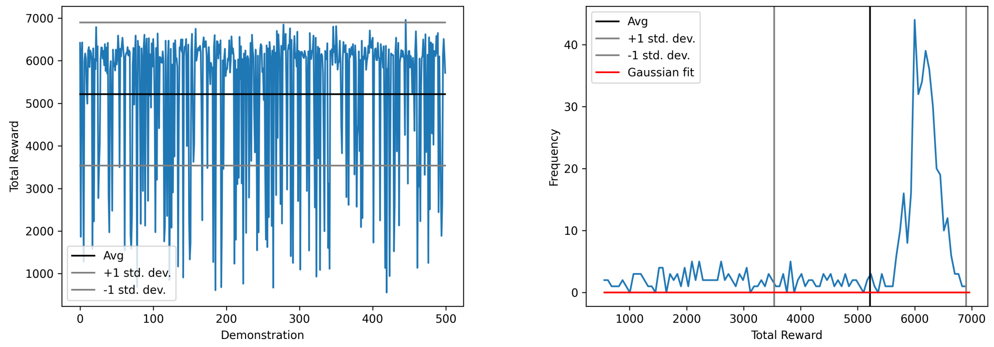


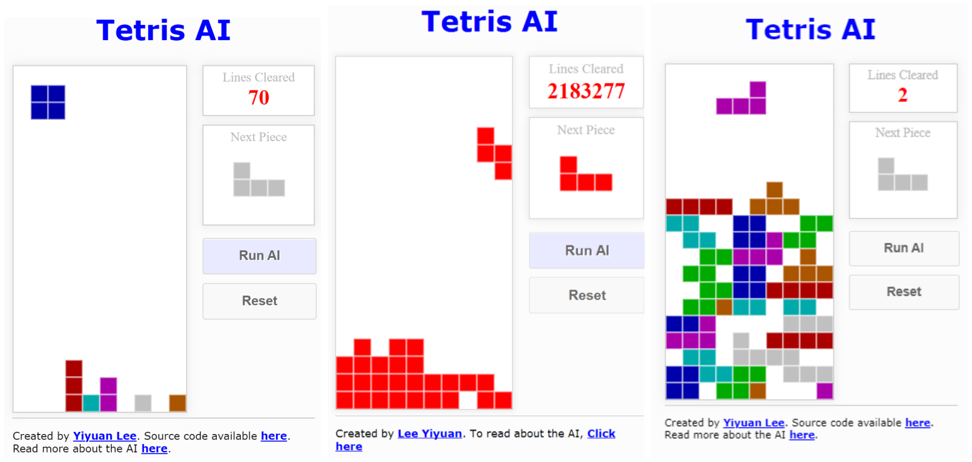
Figure 3: Distribution of Total Reward from Expert Demonstrations

At first glance, it may look like the online expert was better than the human experts, however, the online expert was operating in a *grouped actions* environment, meaning that they were able to perform multiple actions in a single move. When normalized by the number of pieces placed, human experts actually achieve a reward per piece that is five times higher than that of the online expert. This is likely due to the fact that most Tetris experts are trained for survival/longevity and not maximizing reward in a limited time environment. This comparison highlights the complexity of the environment and the importance of carefully interpreting performance metrics.

5 Aside: Grouped Actions Environment

As mentioned before, most online experts [1-4] operate in a *grouped actions* environment, allowing them to perform multiple actions at once. For a game like Tetris, the real challenge is deciding where to place the piece, not how to place the piece. By operating in an *ungrouped actions environment* we complicated the problem by demanding how to place the piece, making Tetris a substantially more difficult problem. By collecting expert actions from a grouped actions environment we are not only automating the collection of data, but we also explored more states than collecting same number of trajectories by hand. Operating in this grouped actions environment allows for $\sim 5\times$ the number of piece placements, meaning that we are able to experience $\sim 5\times$ more states per trajectory. This fixes our previous problem of state exploration (why we initially wanted more data).

A consequence of operating in a grouped actions environment is that we are no longer selecting best actions, but now selecting most optimal future board states. You can rephrase the problem into a *board evaluation* problem, selecting the most optimal board. They [4] created four heuristics to measure board quality (outline below), then used a genetic algorithm to compute the best weights to assign each heuristics.



Created by Yiyuan Lee. Source code available here.
Read more about the AI here.

Created by Lee Yiyuan. To read about the AI, Click here

Created by Yiyuan Lee. Source code available here.
Read more about the AI here.

Figure 4: Online expert agent at `https://leeyiyuan.github.io/tetrisai/`. Left is Normal Conditions, Middle is a Game After 2 Weeks of Running, Right is Recovering from Garbage Board.

- **Aggregate Height**: The sum of height of each column (*minimize* this).
- **Complete Lines**: The total number of filled in rows in one move (*maximize* this).
- **Holes**: The total number of holes: an empty tile with a filled in tile above (*minimize* this).
- **Bumpiness**: Sum of absolute difference between adjacent column heights (*minimize* this).

In addition to adding these heuristics to the environment, we also added the height of each column to account for any more heuristics left unaccounted for. This allowed us to describe each board state as a tuple of fourteen numbers. This would later describe potential board states, and our model would be to pick the best one. Unfortunately, this took longer than expected because there was a discontinuity between expert actions and the grouped actions environment. There was a perfect bijection to what the expert thought were possible actions, and what the environment said were possible actions *except* for one edge case.

The edge case was that the grouped environment actions wrapper we were using from the OpenAI gym had a bug that preventing it from outputting states where the long bar was placed all the way to the left. In the source code for the gym, there was an off-by-one error requiring the logic for the long bar to be shifted by one to the left. It took us a long time to realize this bug, and a long time to fix it. This is because we could not change the expert policy to account for this (because it is a black box to us), so we had to read the source code, enough to find and fix this bug. We will likely make a pull request to the gym notifying them of the bug.

After this we were able to translate the expert policies actions into Python (it was original in JavaScript). On average the expert policy was able to achieve a total reward for one trajectory of $\sim 20,\,000$ before dying, which equates to $\sim 5,000$ if we limit to 1000 moves. In essence, this led us to change the problem we are trying to solve problem by transforming the input and output of the problem. With this detour, we were able to not only understand the problem better, but also leverage the improved parameters to produce better policies overall.

6 Behavior Cloning Improvement

With our new grouped actions environment now working, we had to slightly change the architecture of M2 in order to handle the new input. As a refresher, the next model (Mark 3/M3), took as input forty fourteen-tuples where each fourteen-tuple contains the fourteen heuristic parameters described above. The output of M3 is an index between 0 and 39 denoting which board it chooses. We trained M3 on the BC data collected from the expert policy (data below). We noticed severe under fitting as shown through the training loss being much higher than the validation loss. In order to fix this, we decided to make a more complex model and reduce dropout layer probability from a half to a third. In addition, to help combat under fitting we decided to make the the data more consistent by removing trajectories where the expert died early on.

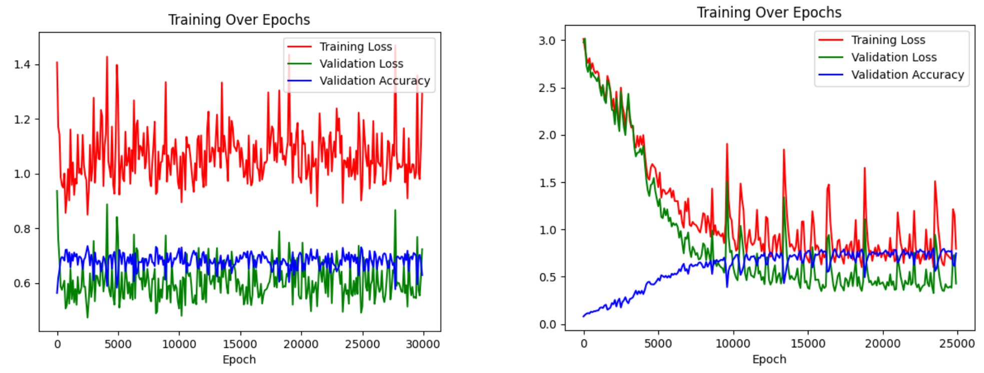


Figure 5: Left is M3 Trained on BC Data. Right is M4 Trained on BC data (with data filtering).

By removing the early death trajectories (died in < 750 moves out of 1000 moves) we reduced the total number of BC trajectories trained on from $\sim 500 \rightarrow\sim 350$. This method showed significant improvement increasing our validation accuracy from $\sim 75\% \rightarrow\sim 83\%$. However, because the model was larger, training this model took ~ 20 hours. The graph above still has not experience the effect of over-fitting, so we anticipate training for longer could get even better results. Unfortunately, we did not think it was worth the time, so we cut training short here.

7 DAgger Improvement

We set up three experiment trainings with DAgger using our M4 model. We trained M4 with a random initialization, with a BC initialization, and lastly with a BC initialization and BC data. Our first run, on the M4 model with BC initialization and BC data we realized that this run would take $\sim 50+$ hours to finish due to additional data and model complexity. Thus in order to make training more reasonable, we cut our BC data in half.

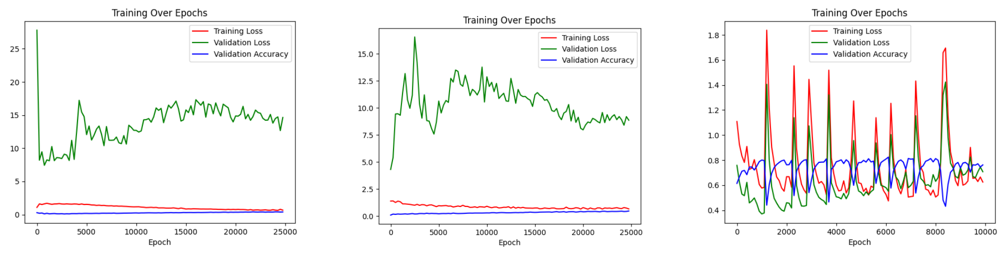


Figure 6: DAgger - M4 with Random Init, M4 with BC Init, M4 with BC Init and Data.

Pure DAgger resulted in severe over-fitting as demonstrated in Figure 6. Completely random initialization has average reward of 20. BC initialization has an average reward of 68. BC initialization (with BC data to start with instead of empty dataset) has an average reward of 193. Because overfitting was such a problem we had to be sure to balance the amount of new data as we are training. The last iteration (far right Figure 6), showcases this perfectly each time the model was beginning to overfit, new expert corrections were issued keeping the model in a good equilibrium state of learning.

8 REINFORCE Improvement

After achieving respectable performance on BC and DAgger, we finally decided to attempt REINFORCE training again using these expert initializations. We expected to get significantly better results than our initial reward threshold of -90, so we were immensely surprised when all three initialization: random, BC, Dagger ended up achieving similar performance to the results in section 3 after 100 epochs of training. We were stuck with these results for a while, and assumed that our project was bust, until we noticed that REINFORCE hit a training loss of 0 within $2/3$ epochs. This is a sign of serious overfitting, and even more so didn't make any sense because we assumed Tetris was too hard of an environment to achieve perfect overfitting in such little time.

We tried a variety of fixes including clipping the gradient to prevent exploding gradients, normalizing the "reward to go", saving on the highest reward instead of the lowest loss, and reducing the learning rate from $5 \cdot 10^{-3}$ to 10^{-8}. These fixes held back perfect overfitting for around $3x$ longer to $8 - 10$ epochs. However, it didn't fully solve our problem. After manually simulating a gradient descent step, we realized that if Numpy or Torch receives a logit large in magnitude (i.e. 1000) then when calculating the softmax probability it defaulted to giving actions a probability of 1. Since our NN didn't actively prevent large logits, we ended up with every action having a probability of 1. Since our loss term includes the logarithm of the probability, we end up with the loss being 0 as $\log(1) = 0$.

To prevent this, we clamped our logits within the range of $[-10, 10]$ while sampling the next action. This prevented the above behavior, but it was also slightly non optimal because logits of 100 and 1000 would both be mapped to 10 and considered the same. Ideally, we would find an activation function that shifts everything instead of just the outliers. However, the ones we could think of are "sigmoid" or "tanh" limit all logits to the range $[-1, 1]$ and $[0, 1]$ respectively. This means that the maximum probability difference between a bad action and a good action is a scale of $e^2 \sim 9$ and $e \sim 3$ respectively, meaning that even if the model is certain of the best action, the worst action is still likely to happen. Training with the above changes never achieves 0 training loss despite getting close, but still isn't the most consistent.

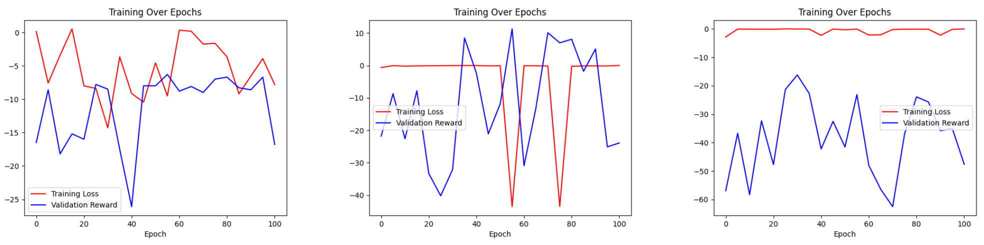


Figure 7: REINFORCE - M4 with Random Init, M4 with BC Init, M4 with DAgger Init.

We were disappointed by the fact that validation reward was mostly negative (with slight peaks above 0). This seemed unexpected as BC and DAgger both achieved rewards above 100 by themselves. We soon realized that this difference was caused by clamping. While clamping was necessary in the training process, it didn't actually prevent the logits from being large, so if we unclamped while evaluating we got much better reward performance as the model was more certain in the moves it made. The average rewards for these models were $-7.68, 108.31, 197.94$ respectively, which is much closer to the results achieved by just BC and DAgger in previous sections.

9 Averaging REINFORCE

While training we noticed a trend when giving REINFORCE a good initialization like BC or DAgger. Specifically, in the first epoch there would be a large decrease in reward (from $100+$ to below 0), in following epochs the reward would slowly increase back up similar to how REINFORCE with random initializations behaved. We interpreted this as the gradient being large initially leading to a large step away from our initialization strategy. We know that our initialization strategies are usually better than what REINFORCE can find on its own (from above testing of random init REINFORCE), so we felt that REINFORCE taking such a big step away from its initialization was causing it to not make use of it properly.

Basically, we theorized that we did not need to promote exploration while training REINFORCE, instead we realized that an attempt to promote conservativeness might lead to better results. While this jumps away from our initial hypothesis, we felt that the information gleaned from repeated trainings of REINFORCE outweighed the hypothesis we made off estimates at the beginning of the project. To implement conservativeness we decided to do a weighted average between our current model and the initial model at various points throughout the training. We decided to mix our models together every 5 epochs, with various weights of $0.5, 0.2, 0.1$. We found that 0.2 worked the best for our current setup, indicating that a heavy reliance on the initialization model performed better than not relying on it at all.

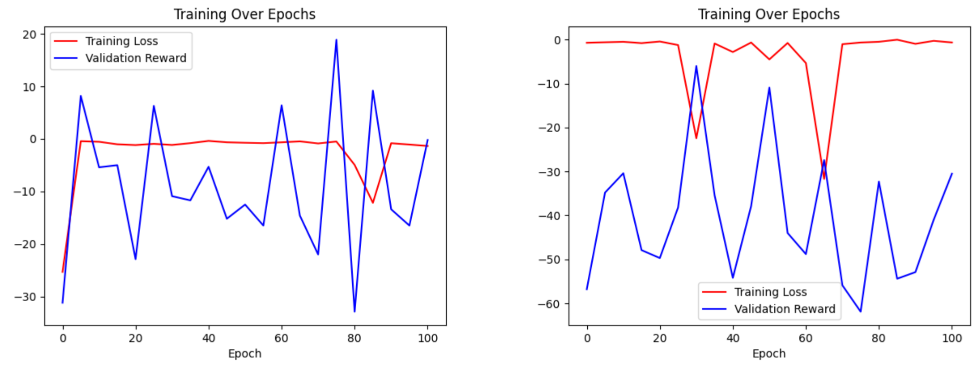


Figure 8: Average REINFORCE (0.2 new weight) - M4 with BC init, M4 with DAgger ini.

10 Performance Reflection

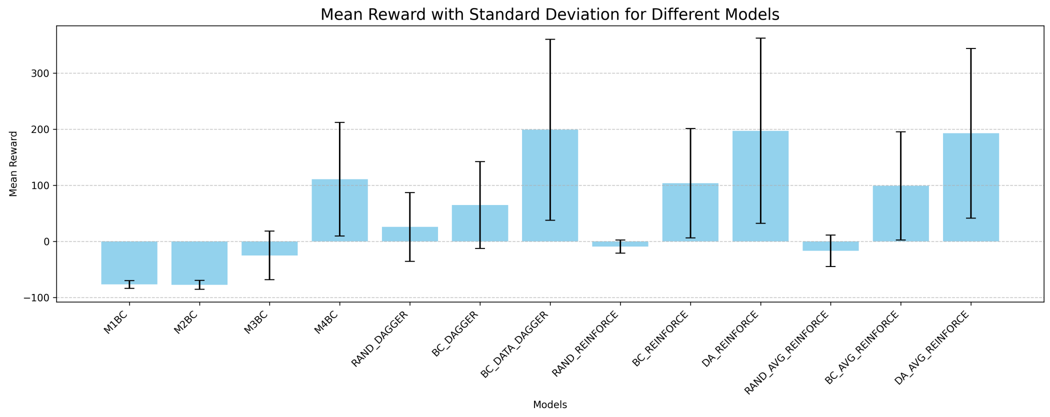


Figure 9: Average reward and variance tracked across 300 demonstrations each

Taking a look at the above graph we can see how our results held up with our hypothesis, predictions, and intuitions. As we can see our initial model architecture improved from M1 architecture to M4 (first 4 bars). The changes we made across these models were increasing model complexity, filtering low performing trajectories, and training with more data. This matches our intuition to increase model performance because it was under fitting to begin with.

Our intuition from class tells us that more descriptive initializations of agents can help algorithms perform better. We have seen this hold true twice in our project. Once for various initialization fed into DAgger, where feeding in BC with more data helped increase performance compared to normal BC and random initialization (shown in the next 3 bars). We also saw this again for various initializations fed into REINFORCE, where feeding in the more descriptive DAgger led to better results than BC or random initialization (shown in the next 3 bars).

So far the results have matched the hypothesis/intuition that we discussed at the beginning of this paper. However, the next result was unexpected. We initially estimated that we needed to add an exploration term, but we realized that conservativeness was likely the better option for performance. The graph above shows that this averaging to be conservative slightly edges out regular REINFORCE but not significant enough to be an amazing result. We think that more optimization through the parameter space of the weighted average is needed to obtain the statistical significance desired, but we didn't have the time/budget to implement it.

11 Next Steps For Tetris

Our current results highlight several avenues for future improvement and exploration. Although we made progress in building, training, and initializing Tetris RL agents, certain aspects of our methodology and environment warrant further refinement. One idea we thought about is to consider integrating a sequence-based averaging approach during training, rather than relying solely on a single snapshot of an initialization. This would involve averaging with the previous checkpoint instead of the original initialization. This approach could smooth out the performance variance seen in the latter half of the bar graph.

Another thing, we thought about is revise the reward structure and incentives. Despite extensive training, our REINFORCE-based agents struggled to discover and exploit the high rewards associated with clearing lines. One plausible reason is that early trajectories often fail quickly, providing insufficient incentive for the agent to explore beyond a small subset of states. Furthermore, illegal action might provide a loophole for termination that allows trajectories to end without good performance, but by escaping the game-over state and going to the illegal-action state, end up getting comparatively better performance.

Future work could involve reward shaping or a more nuanced reward function to better signal the long-term benefits of certain actions. Alternatively, introducing intermediate rewards — for example, rewarding partial board stabilization or piece placements in favorable positions—might help the policy gradient methods identify paths leading to line clears.

Our efforts on the Tetris project the past few weeks underscore the importance of carefully designing every facet of RL, from initialization strategy and model architecture to environment constraints and reward functions. Throughout the process, we learned a lot about the real-world application of RL, and the types of challenges and obstacles along the way. We enjoyed the content of the class this semester and we hope the lessons taught to us stick for a long time.

12 References & Work Distribution

[1] Stevens, M., Pradhan, S., (2016) Playing Tetris with Deep Reinforcement Learning, `https://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf`

[2] Nguyen, V. (2020) Tetris-deep-Q-learning-pytorch, `https://github.com/vietnh1009/Tetris-deep-Q-learning-pytorch/tree/master`

[3] We, M. (2024) Tetris-Gymnasium, `https://github.com/Max-We/Tetris-Gymnasium`

[4] Lee, Y. (2013) Tetris AI – The (Near) Perfect Bot, `https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`

All members agree: There was an equal work distribution across all members.

Mohammad Khan: Created reinforce algorithms implementations, researched expert policies, debugged environment implementation, draft final paper, collected and organized video recordings, helped train the models, wrote human BC data collection scripts.

Pablo Raigoza: Created DAgger algorithm implementations, collected human expert demonstrations, computed most statics used (graphs, numbers, etc.), researched expert policies, automated data collection for BC, draft final paper, helped train the models.

Shao Stassen: Created BC algorithnn implementations, collected human expert demonstrations, video editor, created useful scripts throughout process such as replay functionality, and testing functionality, organized meetings.