ECE 271C: Dynamic Programming
Tetris Project

Connor Hughes

June 7, 2021

Background

Dynamic Programming Basics

Dynamic Programming is a powerful technique which can be used to solve a wide variety of staged optimization problems. Staged optimization problems are those in which a series of decisions are made, which can be thought of as making one decision at each "stage" of the problem, wherein the objective is to optimize some total cost or reward function over all of the stages. At its core, Dynamic Programming leverages the "Principle of Optimality" to efficiently find the optimal solution to staged optimization problems, by breaking them down into smaller subproblems which can be solved sequentially. The Principle of Optimality, originally developed by Richard Bellman, can be stated as follows: an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The Principle of Optimality lends itself to two main implementations of the basic Dynamic Programming algorithm, known as "Forward DP" and "Reverse DP". Reverse DP utilizes the Principle of Optimality exactly as stated above, observing that when seeking optimal solutions which take us from each state at stage k to some terminal state at the end of the sequence of stages, we need only consider the possible transitions and associated costs/rewards to get from stage k to stage $k+1$, **if** we have already solved the subproblem which gives us the optimal solutions to get from each state at stage $k+1$ to the terminal state. Thus, we may begin from the final stage, and efficiently work backwards to the initial state, constructing the optimal solution as we go. Conversely, Forward DP takes advantage of the Principle of Optimality as stated above, but for the flipped problem. That is, Forward DP observes that for any state at stage k, we need only consider the possible transitions and associated costs/rewards to get to that state from stage $k-1$, **if** we have already solved the subproblem which gives us the optimal solution to get from the initial state to each state at stage $k-1$. Thus, we may begin from the initial state, and efficiently work forward through the stages until we reach the terminal state, constructing the optimal solution along the way. The process of determining the optimal value of the cost-to-go (or reward-to-go) by either forward- or reverse-DP is called "value iteration".

One particularly simple problem setting for Dynamic Programming, which serves as a useful example for the approach, is the shortest path problem. In this scenario, states can be visualized as nodes in a network, and the "cost" to go from one state to another can be though of as the "distance" between nodes. In the case where we have a single initial state and a single terminal state, this corresponds to having a predetermined starting and ending node. Within this setting, Dynamic Programming can be used to determine the shortest path between the starting and ending nodes. As this visualization is helpful for understanding the algorithm, the transition between various states across a number of stages will often be referred to as the "path" between those states. However, numerous other types of problems can be formulated as staged optimization problems, and the approach can be extended to handle quite complex situations, as will be shown in the following sections.

When using Dynamic Programming to solve staged optimization problems, often the most challenging part is to first formulate the problem appropriately. To "formulate" a problem for Dynamic Programming means to establish certain definitions which ensure that the Dynamic Programming algorithm can be used, and that it will yield the solution to the desired optimization. These definitions are the following:

State Definition: We must begin by defining the way that we will represent the states which may be encountered at each stage, though there need not be the same set of states nor even the same number of states at every stage. This step is non-trivial, as we typically have numerous options for how to represent a given state, and the choice made here will often impact both the appropriate definitions for the other components of the problem (listed below) as well as the difficulty of its solution using the Dynamic Programming algorithm.

Control: We must also define the way we will represent the decision, also referred to as the "control action", which we may take at each state and each stage.

Set of Admissible Controls: This is the total set of choices from among which we may select a control action. Very often, different states permit different control actions, so this is usually dependent on the current state, though this is not always the case.

Disturbance: In some probabilistic problem settings, there may be an uncontrolled event which impacts the transition from one state to another or the cost/reward associated with that transition. This is modeled as a disturbance term which is included as an input to one or more of the State Transition, Stage Cost, Termination Cost, or Terminal Cost functions (see below). When formulated as such, the disturbance can be factored into the Dynamic Programming algorithm so that it is accounted for in the optimization process. This will be explained in greater detail through the Tetris examples in the following sections.

Stage Cost: This is the cost (or reward) associated with transitioning from a given state at one stage to another state at the next stage. This cost (or reward) may be dependent on the current state, the control action necessary to make the transition, a disturbance term which describes some uncontrolled event, or some combination of these. Dynamic Programming can easily be used to either maximize rewards or minimize costs, and thus we would define either a Stage Cost or Stage Reward accordingly. We can also consider rewards to be “negative costs” and minimize these in order to achieve the desired reward maximization.

Terminal Cost: In some cases, staged optimization problems allow for ending up in one of a number of different states after the final stage. In this setting, it may be necessary to define a “terminal” cost which either rewards or penalizes the final state according to some metric. These can be easily included alongside the stage costs in the DP algorithm and thus included in the optimization process.

Termination Cost: In some settings, our set of admissible controls will include a choice to end the game and accrue no further stage costs/rewards, and this may come with some associated cost or reward. To capture this, we can include a “termination cost” or “termination reward”, which again, can be included alongside the Stage and Terminal Costs and thus be included in the optimization process carried out by the DP algorithm.

State Transition: This function defines the evolution of the state from one stage to the next, and typically depends on the current state, the selected control action, and a disturbance term (when one is included in the problem formulation).

Cost-to-go: This quantity captures the optimal total sum of costs (or rewards) associated with transitioning from a given state at a given stage through the remainder of the stages (or until termination). It thus is the sum of stage costs, terminal cost, and termination cost under a sequence of optimal control actions, and inherently depends on both the current state and the current stage. Storing this quantity for each state at each stage is a central part of the DP algorithm, and the way in which it leverages the Principle of Optimality.

Extensions of Dynamic Programming

As alluded to in the problem formulation laid out above, the Dynamic Programming algorithm can be extended to solve staged optimization problems with stochastic disturbances. In such cases, we have considered optimizing either the worst-case costs-to-go or the expected costs-to-go (wherein we have considered the expected value of the stochastic disturbance in determining the costs-to-go). In addition, under certain assumptions, the Dynamic Programming algorithm can be extended to the case where a problem has an infinite number of stages, which is referred to as an ”infinite-horizon” problem. In the case of infinite-horizon problems, the distinction between subsequent stages disappears, and the goal is to identify an optimal “stationary” policy, which is a policy that is stage-independent.

The key to extending the Dynamic Programming algorithm to infinite-horizon problems is that these infinite-horizon problems must still involve finite costs-to-go. This is intuitive, as we must be able to compare these costs-to-go in order to determine the optimal solution. In order to ensure the costs-to-go are finite, we have considered two cases. The first is where the infinite-horizon problem is analogous to a “stochastic shortest-path” problem, which means that regardless of the control actions taken, after some finite number of stages there is a non-zero probability that we will have reached a termination state. If this can be shown, then it follows that the likelihood of continuing *without* reaching a termination state decreases exponentially with the stage number. If this is the case, and we have finite stage costs, termination costs, and terminal costs, then we know that the expected cost-to-go from any state must be finite. The second is where we consider a "discounted" infinite-horizon problem, in which we weight future costs by a factor which decreases with each stage, which for the right rate of discounting can also ensure finite costs-to-go.

When considering either of these two cases, we observe a few interesting and useful properties. First, the optimal expected cost-to-go or reward-to-go which is determined by value iteration over a finite number of stages converges to the correct value for the infinite-horizon case as the number of stages used in the value iteration process tends to infinity. This means that we can run value iteration with an increasing number of stages, and simply look for convergence of the expected costs-to-go (or rewards-to-go) in order to find the value of these quantities for the infinite-horizon case. Second, this convergence means that the optimal stationary policy and the expected value of the cost-to-go for the infinite-horizon case satisfy the fixed-point equation which is the optimization of the recursive cost-to-go update equation used in the value iteration process (more discussion on this topic is included in the infinite-horizon Tetris sections which follow). This fixed point equation is known as the Bellman Equation, and it can be solved directly (though doing so is often computationally expensive), which gives us another means of determining the cost-to-go for the infinite horizon case.

Furthermore, we can also consider the fixed-point equation given by the optimization of the recursive cost-to-go update equation used in the value iteration process, but in the case where we only have one admissible control policy. In this situation, the expression is known as the "μ-specific Bellman Equation". This expression can then be used to implement a process known as "policy iteration" wherein we update the control policy based on the optimization of the value iteration recursion wherein we use the current control policy and the μ-specific Bellman Equation to compute the cost-to-go from the next stage. More information on this is included in the following sections on infinite-horizon Tetris.

Lastly, we also consider staged optimization problems in which the number of states or the number of admissible controls (or both) is so large that it makes standard Dynamic Programming intractable due to the required number of computations, which is referred to as "the curse of dimensionality". In these cases, we are still able to determine effective (though sub-optimal) control policies using a powerful technique called Approximate Dynamic Programming. Approximate Dynamic Programming is a form of Reinforcement Learning, and involves using some type of function which does not rely on the full state representation to approximate the cost-to-go from a given state. The aim is to leverage the Principle of Optimality to improve the approximation of the costs-to-go. The justification for this approach is that, if we have a function which approximates the cost-to-go well, then the optimal control policies which we determine on the basis of this approximation should also perform well in practice. This approach is discussed further in Section 5.

Tetris Problems

As explained in the section above, Dynamic Programming is an extremely useful technique for solving staged optimization problems, and it can be extended to handle a wide variety of problem settings. The game of Tetris provides a compelling environment in which to explore these possibilities, as it is naturally formulated as a staged optimization problem, and can easily be made simpler or more complex by altering the size of the Tetris game board, the allowed number of moves, and the sequence in which pieces which are presented. The following sections provide an excellent overview of the ways the Dynamic Programming methodology can be applied (and extended) to determine optimal (or near-optimal) piece placement within numerous variations of the game of Tetris.

1 Deterministic Tetris - Finite Horizon (3x3)

We begin by considering a greatly simplified version of the traditional game of Tetris, which involves only a 3x3 game board, 3 game pieces, a deterministic sequence of appearance of those pieces, and a maximum of 100 moves. This setup allows for us to apply the standard DP algorithm to determine the optimal piece placement at each stage. The game board and set of pieces are shown in Figure 1 below:

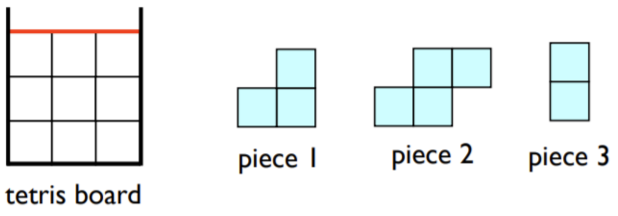

Figure 1: Game board and pieces used for finite horizon, deterministic 3x3 Tetris.

We note that every configuration of the Tetris game board involves some combination of filled and unfilled squares. That is, squares where there either is or is not a piece. So, we can uniquely identify all of the possible board configurations with a 9-bit binary word representation, where each bit corresponds to a single square, a value of 1 corresponds to the square being filled, and a value of 0 corresponds to the square being unfilled. While some of these board configurations are not actually reachable by any sequence of piece placements starting with an empty board, this is not a concern as the total number of board configurations is only $2^9 = 512$, and thus it is computationally tractable to consider all board configurations. We can ensure that the optimal solution we determine by the DP algorithm adheres to the rules of Tetris by assigning an infinite Stage Cost to transitions which are impossible.

However, to fully describe the state, we must also include the current piece which is available to be placed, which gives 1536 possible states. Lastly, we also consider one "gameover" state which corresponds to a board configuration wherein the maximum height of one or more columns exceeds the height limit (3 rows) and thus the game has ended. This brings the total number of states to 1537. For convenience, we will store board configurations not as a 9-bit binary word, but as its decimal equivalent. We can then complete the formulation of this problem for Dynamic Programming by defining the following:

State Definition: $x_k \in \{(B, PN), EOG\}$ where B is the decimal representation of the board configuration, PN is the current piece number, and EOG is the additional state which corresponds to "end-of-game".

Control: u_k, the orientation and position in which the current piece is placed.

Set of Admissible Controls: $U_k(x_k)$, the set of all orientations and positions in which the current piece can be placed, which of course depends on the current board configuration and piece.

Disturbance: None.

Stage Reward: $g(x_k, u_k) = NR$ where NR represents the number of rows eliminated by the chosen move. The Stage Reward for transition from the "end of game" state is defined to be zero.

Terminal Cost: None.

Termination Cost: None.

State Transition: $x_{k+1} = f(x_k, u_k)$. The State Transition encodes the rules of Tetris, updating the board configuration appropriately, eliminating rows when applicable, and supplying a new piece for the next stage. The transition from the "end of game" state is automatically set to go to the "end of game" state again at the next stage.

Reward-to-go: $J_k(x_k)$. In accordance with the above definitions, the cost-to-go from a current state at a given stage is the number of rows which can be eliminated in the remaining stages, starting with the current board configuration and available piece.

This problem was solved using reverse DP, for the case where the number of stages (i.e., the number of moves) was limited to 100. The pieces are presented in the predetermined order $\{1, 2, 3, 1, 2, 3, ...\}$, although we shift this twice so that we can consider the cases where we begin with pieces 1, 2, and 3. This means that we do not need to consider all 1537 permissible states at each stage when we carry out the DP algorithm, rather, we need only consider the 512 board configurations corresponding to the known next piece, as well as the "end of game" state. In each case, the initial board configuration is an empty board. The results suggest that the DP algorithm was able to successfully learn the optimal piece placement for each stage, and eliminate a large number of rows. The results are summarized in the table below:

Starting Piece Number	Number of Rows Eliminated
1	100
2	100
3	99

2 Worst-Case Tetris - Finite Horizon (3x3)

The next variation on the standard Dynamic Programming algorithm which we explored was to suppose that the sequence of pieces was unknown, and to consider the worst-case scenario for the next piece when determining the optimal policy. The aim of this approach is to develop a robust policy which mitigates how poorly the game of Tetris is played under any circumstances, rather than to optimize how it is played under known circumstances (that is, with a known sequence of pieces). One defining feature of such situations is that we must choose our control action prior to learning what the next piece will be. While this may not be the best approach for Tetris, in staged optimization scenarios which involve an intelligent adversary (and a corresponding "Adversarial Disturbance"), this approach is clearly appropriate. In such situations, by maximizing performance under the assumption that the new situation is always made as unfavorable as possible, the worst-case Dynamic Programming approach enables the development of a strategy which is highly effective against an opponent.

More formally, in the presence of a disturbance w_k, the aim of worst-case Dynamic Programming is to identify the set of policies π which satisfies:

$$\min_{\pi}(J_{\pi}(x_0)) = \min_{\pi}(\max_{w_0,w_1,\ldots,w_N}(g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)))$$

where

$$\pi = \mu_1, \mu_2, ..., \mu_{N-1}$$

is the set of policies which determine the control action at each stage. In the case of Tetris, since the aim is to maximize reward (rather than minimize cost) the minimization and maximization above should be switched, and the appropriate expression can be written as follows:

$$\max_{\pi}(J_{\pi}(x_0)) = \max_{\pi}(\min_{w_0,w_1,\ldots,w_N}(g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)))$$

Furthermore, in Tetris, since there is no Terminal Cost, $g_N(x_N) = 0$, and the Stage Cost function has no dependence on the stage number, we see that $g_k(x_k, \mu_k(x_k), w_k))$ can be written as $g(x_k, \mu_k(x_k), w_k))$.

As mentioned above, when determining π, we must evaluate the costs-to-go from each state at each stage under the assumption that we will receive the worst piece at the next stage. In the context of 3x3 Tetris, we can use this approach to compute the costs-to-go from an empty board, with each of the 3 possible pieces, using a worst-case analysis. This process yielded the following results:

Starting Piece Number	Number of Rows Eliminated
1	2
2	2
3	1

Clearly, the optimal performance is dramatically worse in the worst-case scenario than in the scenario with a deterministic sequence of pieces considered in the previous section. From a different perspective, though, we notice that this approach enables us to find a policy which ensures that we always eliminate a row, regardless of the order in which pieces appear. For most sequences of pieces, the performance which is obtained using the policy determined by worst-case Dynamic Programming should be appreciably better than the worst-case results. To see this, we applied the policies determined by worst-case analysis to the scenario where we receive the deterministic sequences of pieces described in the previous section, which yielded the following results:

Starting Piece Number	Number of Rows Eliminated
1	6
2	2
3	99

As we can see, the performance is improved over the worst case scenario when we start with piece 1, and dramatically improved when we start with piece 3, but remains the same for piece 2. We can verify that this result is accurate by examining the pieces (shown in the previous section) and noticing that the sequence 2-3-1-2-3 does not allow for eliminating more than 2 rows. Furthermore, we can conclude that worst-case analysis is a useful extension of Dynamic Programming, particularly in situations with an intelligent adversary, but that it also has the potential to significantly hinder performance in other situations where the worst-case event does not often occur.

3 Stochastic Tetris - Finite Horizon (3x3)

Next, we consider the case where the occurrence of new pieces in the Tetris game is stochastic. In this setting, we must alter the way in which we have formulated the problem for Dynamic Progamming, by introducing the following definitions:

Disturbance: w_k, the piece which is encountered at stage $k+1$

State Transition: $x_{k+1} = f(x_k, u_k, w_k)$. The State Transition still encodes the rules of Tetris, updating the board configuration appropriately, eliminating rows when applicable, and supplying a new piece for the next stage. The transition from the "end of game" state is still automatically set to go to the "end of game" state again at the next stage. However, now the new piece for the next stage is given by w_k.

Reward-to-go: $J_k(x_k) = g(x_k, \mu_k(x_k)) + \mathbb{E}_{w_k}[J_{k+1}(f(x_k, \mu_k(x_k), w_k))]$. The Reward-to-go is now evaluated by taking the expectation over the disturbance w_k of the Rewards-to-go from the possible next states. This enables us to construct the set of policies π which optimizes the expected number of rows eliminated in the presence of stochastic occurrence of new pieces.

We can consider the situation wherein the occurrence of the next piece is stochastic but depends upon the current piece, with a known set of transition probabilities between pieces at successive stages. This approach was used in order to develop a more flexible DP algorithm, and in order to verify that the algorithm performed correctly, as it is easy to use such a setup to simulate the deterministic sequences of pieces discussed in the previous sections. However, when considering actual stochastic behavior, we have considered only the case where each piece occurs at the next stage with equal probability, for any current piece. Solving this problem for a Tetris game with a 3x3 board, a maximum of 100 moves, and the stochastic sequence of pieces described, using reverse DP with the Reward-to-go definition given above, yields the following results:

Starting Piece Number	Expected Number of Rows Eliminated
1	14.91
2	10.14
3	14.39

Then, we can verify these results are accurate by running numerous simulated games of Tetris, beginning with each of the three pieces, where the pieces occur with equal probability at each stage and the control actions are selected in accordance with the policies we have found which maximize the expected number of rows eliminated. After a sufficient number of games, the average number of rows eliminated in simulation converges to a value close to the expected values given in the above table. This can be seen from the plots below, which show the number of rows eliminated per game as well as a running average of that value across a series of 2000 games, using the policies described:

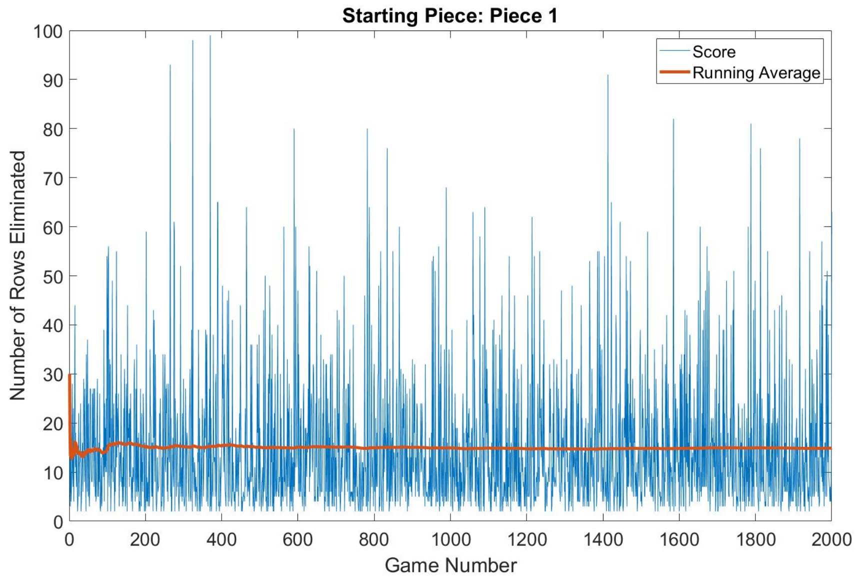

Figure 2: Number of rows eliminated across 2000 simulated games of stochastic 3x3 Tetris, each of which begin with an empty board and Piece 1.

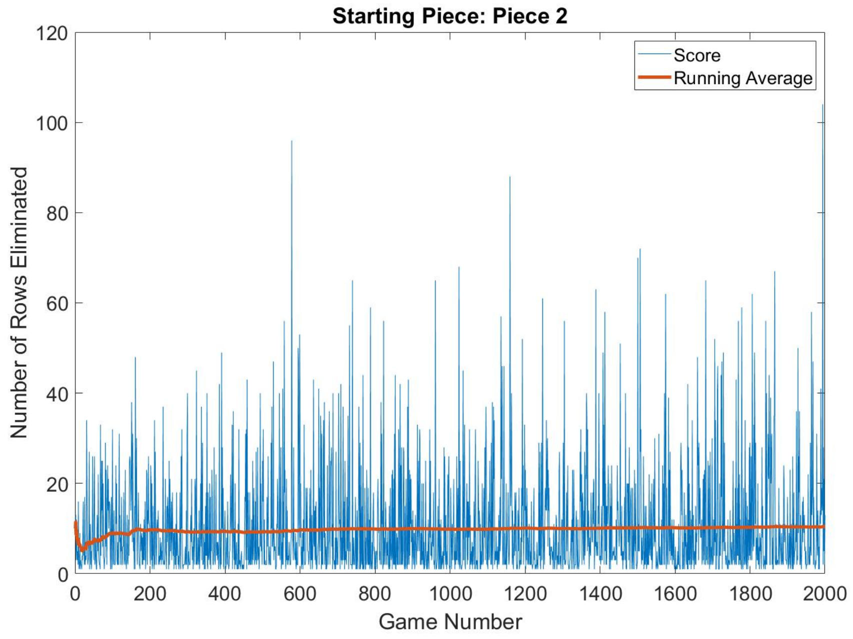

Figure 3: Number of rows eliminated across 2000 simulated games of stochastic 3x3 Tetris, each of which begin with an empty board and Piece 2.

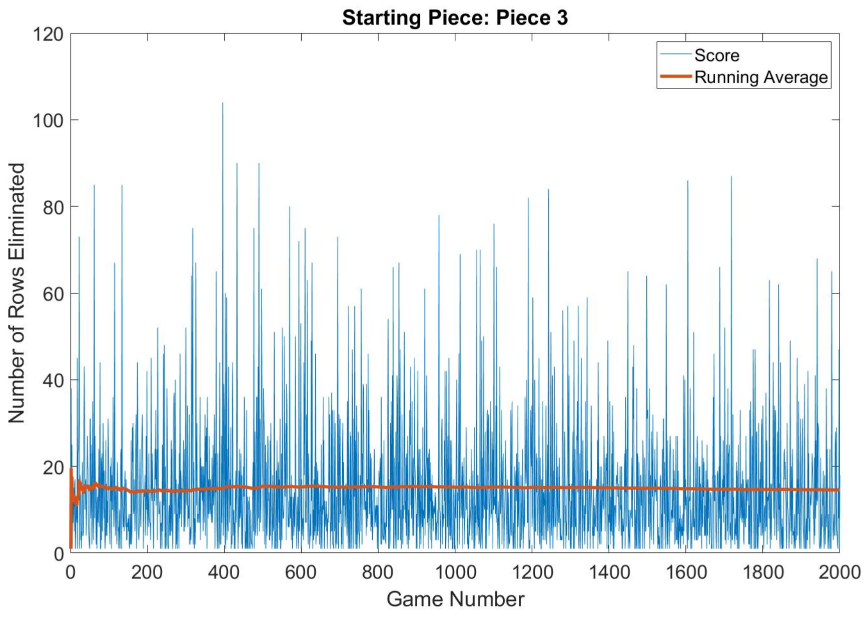

Figure 4: Number of rows eliminated across 2000 simulated games of stochastic 3x3 Tetris, each of which begin with an empty board and Piece 3.

The following table summarizes these empirical results:

Starting Piece Number	Average Number of Rows Eliminated
1	14.8905
2	10.4415
3	14.5640

4 Stochastic Tetris - Infinite Horizon (3x3)

Next, we begin to explore the situation wherein there is no limit on the number of moves per game, which is to say we shall consider the infinite horizon case. For now, we shall still consider only a 3x3 game board. As mentioned in the "Extensions of Dynamic Programming" section previously, when we begin to consider the infinite horizon case, we must verify that our problem formulation will give finite expected rewards-to-go from all states, under all admissible control policies. This can be done by showing that regardless of the control policy chosen, there exists a sequence of pieces which will cause the game to terminate in a finite number of moves.

With only a 3x3 game board and 3 pieces, we can find such a sequence by playing a few games by hand. For example, we can see that it is not possible to continue to play more than a few moves if we repeatedly receive Piece 2 (the 'S'-shaped piece). Thus, we can see that in the case where we receive a stochastic selection of pieces, the probability of playing without encountering such a sequence decreases exponentially with the number of moves made. So, since we have finite stage rewards (the number of rows eliminated by any given move), and an exponentially decreasing probability of continuing to play the game, the expected number of rows eliminated over an infinite number of moves will also be finite. That is to say, stochastic Tetris is analogous to a stochastic shortest path problem.

Naturally, we might expect that as the number of stages we are considering grows to become infinite, the distinction between subsequent stages vanishes. Accordingly, we do not seek an infinite sequence of policies which will provide the optimal solution for 3x3 Tetris. Rather, in the infinite horizon case, we seek an optimal "stationary" (or stage-independent) policy. So, we are no longesr concerned with a sequence of stage-dependent policies $\pi = \{\mu_0, \mu_1, ...\}$, and instead we simply have μ.

To describe the infinite-horizon case in a more intuitive way, we will alter our notation for the stage numbering so that it is reversed. Now, rather than treating stage 0 as the initial state, and taking some finite stage number N to be the final state, we will instead take 0 to be the final state, and increment the stage numbers "backward in time" away from this final state. This new notation gives the following new form of the stochastic-case reward-to-go recursion given previously:

$$\begin{aligned} J_{k+1}(x_{k+1}) &= \max_{\mu}\big(\mathop{\mathbb{E}}_{w_{k+1}}[g(x_{k+1}, \mu(x_{k+1})) + J_k(x_k)]\big) \\ &= \max_{\mu}\big(\mathop{\mathbb{E}}_{w_{k+1}}[g(x_{k+1}, \mu(x_{k+1})) + J_k(f(x_{k+1}, \mu(x_{k+1}), w_{k+1}))]\big) \end{aligned}$$

We can then see that the optimal stationary policy must maximize the expected reward-to-go, that it must be a fixed point of the recursion written above. We can express this with the following equation, which is known as the Bellman Equation:

$$J^*(x) = \max_{\mu}(\mathop{\mathbb{E}}_{w}[g(x, \mu(x)) + J^*(f(x, \mu(x), w))])$$

Furthermore, it is intuitive, and indeed it can be shown, that the value iteration process for the finite-staged optimization problem converges to the expected number of rows eliminated for the infinite-horizon case, as the number of stages in the finite-staged optimization is increased. Thus, we can increase the number of stages in the value iteration process used in the previous section in order to see the convergence in the expected rewards-to-go, and we know that these will correspond to the expected number of rows eliminated in the infinite horizon case. This process was carried out, and the rewards-to-go are plotted below as the maximum number of moves (i.e., the number of stages) used in the value iteration process was increased from 1 to 200.

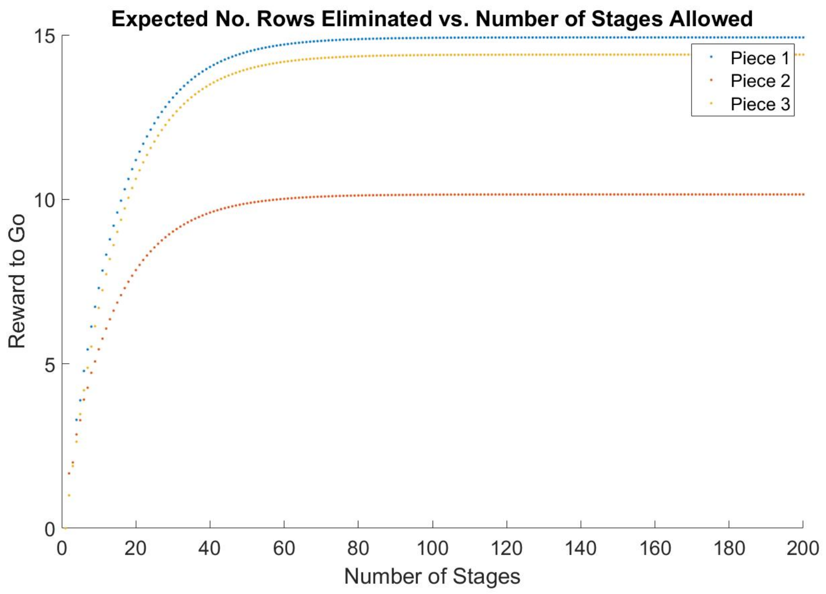

Figure 5: The rewards-to-go from an empty board starting with each piece are plotted against the number of stages used in the value iteration process, which corresponds to the maximum number of moves allowed.

Clearly, we can see from the plot above that the expected rewards-to-go are converging. The final values to which each converged in the 200-stage case are given in the table below:

Starting Piece Number	Expected Number of Rows Eliminated
1	14.9229
2	10.1486
3	14.4008

Furthermore, we find that as the number of stages used in the value iteration process increases, the optimal control actions which are found for the highest stage numbers will converge to the optimal stationary policy. This means that using the value iteration process above for an increasing number of stages, we have obtained a good approximation for the full optimal solution to the infinite-horizon problem. Another approach would have been to use policy iteration to determine the optimal stationary policy, which utilizes the μ-specific Bellman equation, as discussed in the section, "Extensions of Dynamic Programming".

5 Stochastic Tetris - Infinite Horizon (various board sizes)

Lastly, we will consider a significantly more complex game of Tetris, wherein we have stochastic piece selection, an infinite horizon (i.e., no limit to how many moves can be made), 7 pieces, and a game board with 15 rows and 10 columns. This variation on Tetris provides an interesting and useful learning opportunity, as we are forced to confront the curse of dimensionality. Under these conditions, the size of the state space and the set of admissible controls has grown so large that solving the problem by any of the methods used in the previous sections (each of which rely on a full representation of the state, and iteration through every state at every stage) has become entirely intractable due to the number of computations required. We will begin by exploring the performance of a few ad-hoc policies, then we will apply techniques from Approximate Dynamic Programming. In doing so, we will try to leverage the concept of the Principle of Optimality in order to develop a near-optimal policy using an approximation of the reward-to-go from each state (which does not require the full state representation).

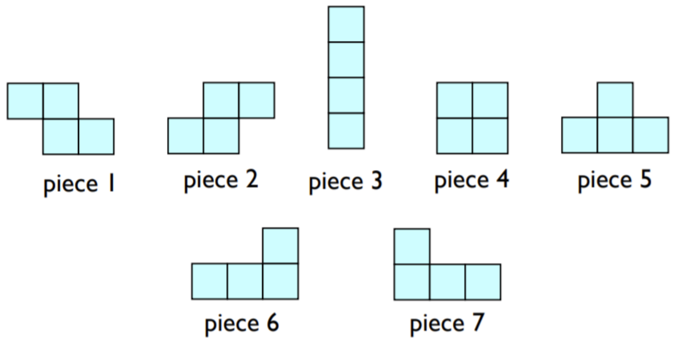

Figure 6: The set of game pieces which will be used in full-scale Tetris.

Part (a): Studies on simplistic ad-hoc policies

We begin our exploration of this variation of Tetris, and thus our confrontation of the curse of dimensionality, with the simplest possible approach. We can develop a few simple policies by hand which seem likely to lead to some degree of success in playing Tetris, then we can simulate these policies over a large number of games to evaluate and compare their performance. The first such ad-hoc policy which was implemented was simply to minimize the resulting maximum column height, measured from the bottom of the game board, when choosing where to place the current piece. This policy was run for 100 games, and the running average of the number of rows eliminated per game was plotted, as shown below:

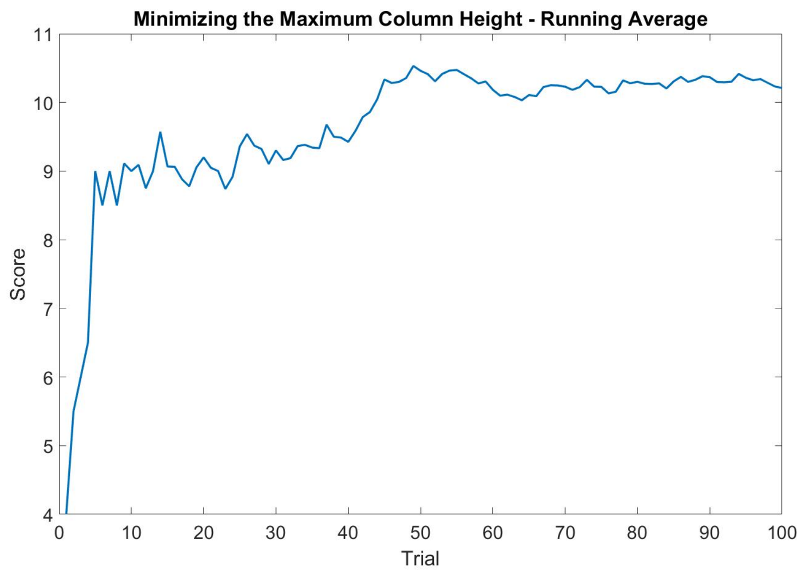

Figure 7: Running average of the number of rows eliminated per game using the ad-hoc policy of minimizing the maximum column height.

As the above plot shows, after 100 games, the average number of rows eliminated with the first ad-hoc policy was approximately 10 rows. The next ad-hoc policy which was implemented was to minimize the resulting average column height when selecting how to place the current piece. This policy was also run for 100 games, and a running average of the performance is shown below:

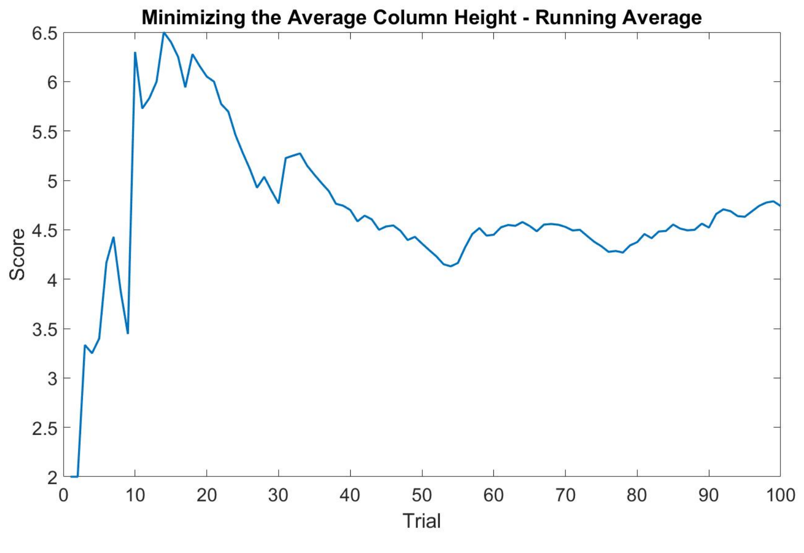

Figure 8: Running average of the number of rows eliminated per game using the ad-hoc policy of minimizing the average column height.

As can be seen from the plot above, this policy was approximately half as effective as minimizing the maximum column height, eliminating an average of 5 rows across 100 games. The next ad-hoc policy which was evaluated was to maximize the average distance between the tops of the columns and the column height limit (15 rows). Once again, this policy was run for 100 games, and a running average of the performance is shown below:

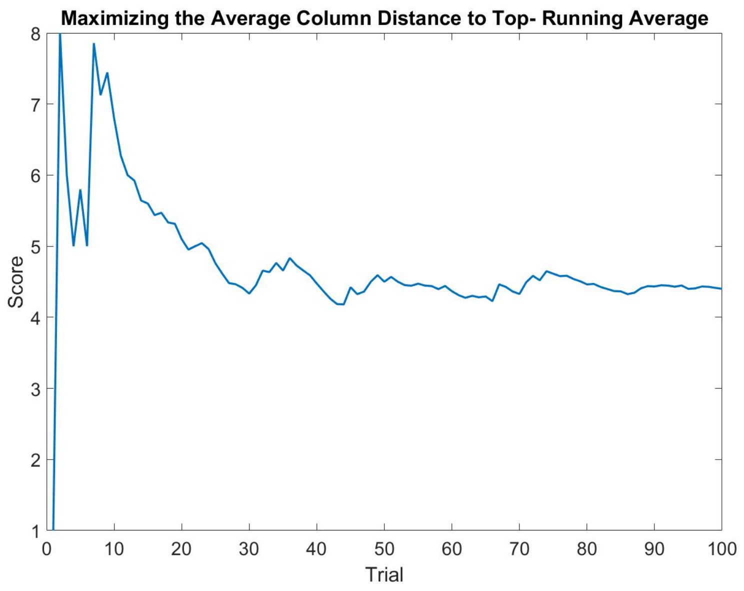

Figure 9: Running average of the number of rows eliminated per game using the ad-hoc policy of maximizing the average distance between the tops of the columns and the height limit.

Not too surprisingly, this last ad-hoc policy performed very similarly on average to the second ad-hoc policy. The second and third ad-hoc policies are based on essentially identical information about the state. While it was an interesting test, it seems that in this case, the difference in formulation does not impact performance. The final ad-hoc policy which was simulated was to minimize the sum of the squares of the column heights. The intent with this policy was to try to incentivize both placing the current piece as low as possible and eliminating rows whenever possible. Once again, this policy was run for 100 games, and a running average of the performance is shown below:

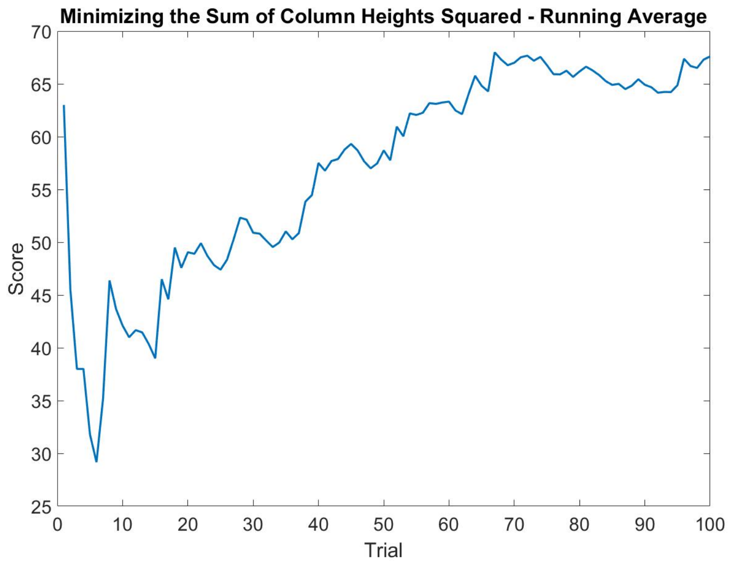

Figure 10: Running average of the number of rows eliminated per game using the ad-hoc policy of minimizing the sum of the column heights squared.

Clearly, the performance is dramatically improved by using this ad-hoc policy, and the average number of rows eliminated across 100 simulated games was approximately 67.

Part (b): Temporal Difference Learning + Basis Selection

The next approach which will be used to try to develop an algorithm which plays full-scale Tetris successfully is a form of Approximate Dynamic Programming (and a form of Reinforcement Learning) called Temporal Difference Learning, often abbreviated TD-learning. Conceptually, TD-learning is quite similar to standard policy iteration, but it involves an approximation of the reward-to-go from each state which does not require the full state representation. This is the means by which TD-learning might help us circumvent the curse of dimensionality.

Standard policy iteration involves a policy update step based on the recursive reward-to-go update equation used in value iteration, which depends upon the current reward-to-go J_μ, which is computed using the μ-specific Bellman Equation, as follows:

$$\mu^+(i) = \arg\min_{u \in U(i)}(g(i,u) + \sum_{j=1}^{n} p_{ij}(\mu(i))J_\mu(j))$$

where

$$J_\mu(i) = g(i,\mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i))J_\mu(j)$$

and i and j are arbitrary states at success stages, with p_{ij} representing the probability of transition from state i to state j.

However, in the case of TD-learning, we substitute a function of the form $r^T\phi(x_k)$ for the reward-to-go J_μ. This function is an approximation of the reward-to-go from the state x_k which is computed using weighted feature vectors. Feature vectors are extracted from a given state, in this case a Tetris board configuration, and they encapsulate information about some aspect of that state. In the previous section, the aspects of the board configurations which were used to make decisions in each of the ad-hoc policies, such as maximum column height, average column height, and so on, could have each been captured a feature vector. The vector of weights r is used to try to approximate the reward-to-go from a given state on the basis of the extracted features. In a nutshell, the aim of TD-learning is to use a gradient-descent-like method to try to learn the weights necessary to optimize how well the reward-to-go is approximated. Then, the theory is that a good approximation of the reward-to-go can be used to select a policy which, while not actually optimal, will perform well in achieving the desired objective. So, in addition to the weight-updating process, a policy iteration process is carried out, which allows us to build upon the Principle of Optimality to try to improve our control policy using the approximated reward-to-go function.

However, the policy iteration process is not the only place where we leverage the Principle of Optimality with TD-learning. We also make use of the concept in our weight-updating process. The weight update process involves the computation of a quantity called the Temporal Difference, which is given by the following expression:

$$d_t = g(x_t, u_t) + r_t^T\phi(x_{t+1}) - r_t^T\phi(x_t)$$

where x_{t+1} is the optimal next state as determined by the approximate reward-to-go which is computed using the *current* weights. Then, the weight update step is completed as follows:

$$r_{t+1} = r_t + \gamma_t d_t z_t$$

where

$$z_t = \sum_{\tau=0}^{t}(\alpha\lambda)^{t-\tau}\phi(x_\tau)$$

where d_t is the temporal difference, γ_t is a step-size parameter which is chosen in order to induce convergence in the evolution of the weights over a series of stages, and z_t is a direction vector which determines the way in which the weights should be changed. By changing the parameter λ, the direction vector z_t can be set up to make the weight update in accordance with the feature vectors extracted from the current state, or it can be treated as a decreasingly-weighted sum of the feature vectors from the most recent states, which in effect "shares the responsibility" for the current reward-to-go across the last few states, as opposed to placing it entirely on the most recent state (which is the case when λ, is set to 0, in which case the algorithm is thus called TD(0)).

For simplicity, the first implementation tested was TD(0), with only a few features extracted from each state. The features chosen were the sum of the heights squared (scaled down by a constant factor so that this feature would have approximately the same magnitude as the other features), as well as the number of holes in the wall of bricks, the sum of the absolute value of the differences between adjacent column heights, and the maximum column height. The performance of the TD(0) algorithm was evaluated during the training process by plotting a running average of the number of rows eliminated per game, while the weights are being updated periodically.

There are two versions of the algorithm which were attempted: one in which an update is made after every iteration to the weights which are used to determine the next control action, and one in which there is a timescale separation, in that the same weights are used to determine control actions for a period of time (in this case 10 games) before being updated, while the "new" weights are still iterated using the weight update expression above at every single stage (that is, every single move). In either case, when the actual control action is chosen, this is not done strictly according to the approximate reward-to-go function and the stage cost. Instead, the control choice is made using a "softmax" function, which makes the optimal choice some percentage of the time, and at other times makes a non-optimal choice. This function can be tuned to make various choices with varying probabilities, and to make the optimal choice a higher or lower percentage of the time, using a parameter called the "temperature".

There are numerous hyperparameters which can be tuned to try to achieve the desired learning with TD(λ), including the step size, the rate of decrease of the step size, the initial values for the weights, the set of features being used, the softmax temperature, the frequency of the weight updating, the number of games to run, the value of λ itself, etc. Multiple days were spent experimenting with a whole range of values for each of them, and unfortunately, little success was found. Ultimately, after running dozens and dozens of attempts, the best results which were seen using TD(λ) were obtained using TD(0), and they showed what appears to be brief improvement in gameplay followed by significant decline, as shown in the figures below:

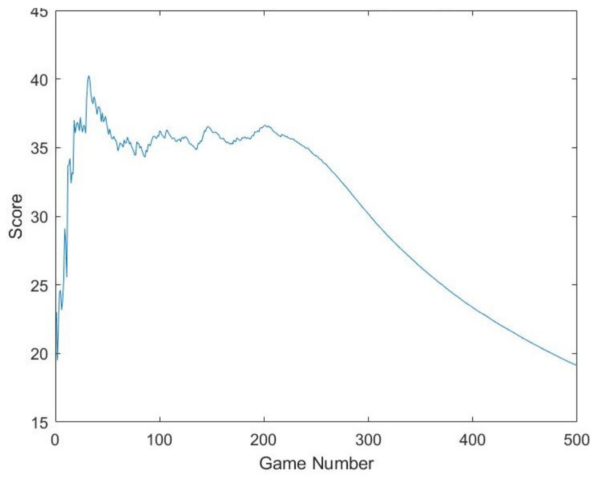

Figure 11: Running average of the number of rows eliminated per game using a TD(0) algorithm training over 500 games.

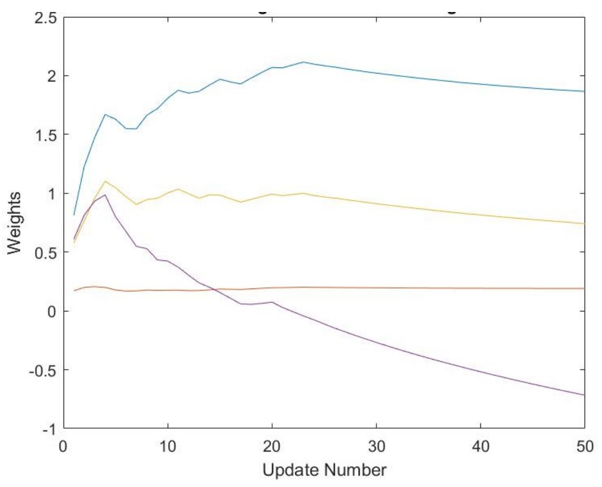

Figure 12: Evolution of the feature vector weights using a TD(0) algorithm training over 500 games.

As can be seen from the above plot of the weights, the weights did not quite fully converge in the number of updates shown. However, there were other trials run where the weights did converge, and where a greater number of games were used in the training process, with similar or worse performance than shown above for the number of rows eliminated.

In seeking to understand this apparent “learning and then un-learning” phenomenon that was observed, we turned to Benjamin van Roy’s book chapter “Neuro-Dynamic Programming: Overview and Recent Trends”. This book chapter was found in the form of a paper as well, and the paper referenced another work by Dmitri Bertsekas and Sergey Ioffe titled “Temporal Differences-Based Policy Iteration and Applications in Neuro-Dynamic Programming”, which happened to include an attempted application of TD-learning to the game of Tetris, as well as the use of another method called λ-Policy Iteration for the Tetris problem. Interestingly, Bertsekas saw the exact same “learning and unlearning” phenomenon. Bertsekas commented that the TD(λ) approach ran into “serious difficulties” and he ultimately abandoned his attempt to use TD-learning for the Tetris problem, concluding that the failure of the algorithm could be attributed either to large-scale simulation noise or to “the ill-conditioning to which gradient-like methods such as TD(λ) are susceptible”.

Following in Bertsekas’ footsteps, in part (d) below we attempt (successfully) to implement the λ-policy iteration method which Bertsekas found far more successful than TD(λ).

Part (c): Policy evaluation through simulated play

The best simulated play of the game of Tetris using the weights learned with the TD(0) algorithm is shown as a plot of the running average over 100 games below:

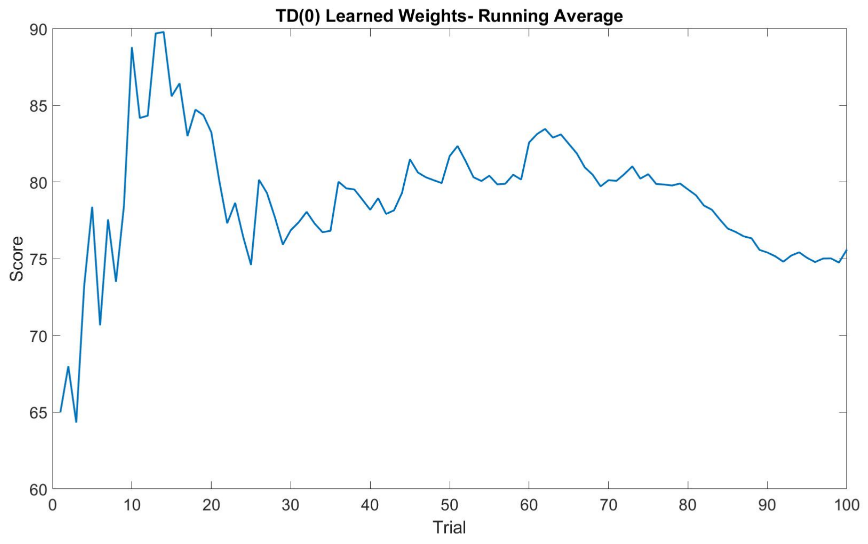

Figure 13: Performance in Tetris gameplay using weights learned with a TD(0) algorithm trained over 500 games.

From the above figure, we can see that the performance is not too bad, with an average of about 80 rows eliminated per game over the course of 100 games. The more concerning sign regarding the performance of the TD(0) algorithm was the "learning and then un-learning" phenomenon previously described.

Part (d): Alternative methods

Lastly, we implemented the λ-policy iteration method described in Bertsekas' paper, which yielded much better results than TD-learning for both Bertsekas and for us. The algorithm is somewhat similar to TD-learning, however, the weight update process does not involve a gradient-like method, but rather a direct least-squares optimization. The formula which Bertsekas provides (equation (4.1) in the paper mentioned above) can be reduced by setting λ equal to zero, which has much the same effect in this case that it does with TD-learning, as was described in part (b). Once that simplification is made, the expression given by Bertsekas reduces to a least squares optimization of the weights to maximize the approximate reward-to-go improvement based on the sum of the feature vectors seen over a chosen number of stages (in our case 10 games). This approach seems to be more stable, and more effective in optimizing the weights given the sparse rewards and potentially ill-conditioned gradient present in the Tetris example. The results of this process are shown in the plots below:

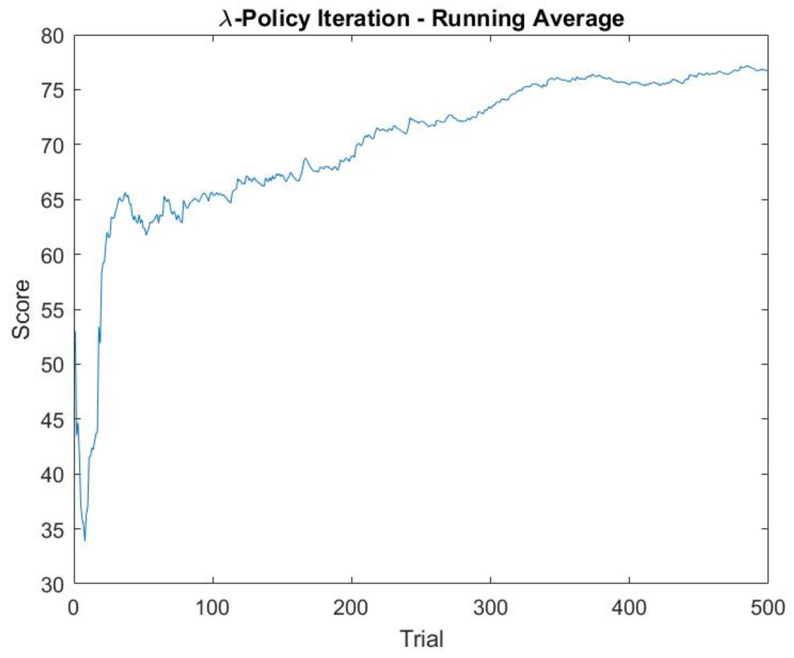

Figure 14: Running average of the number of rows eliminated per game using a λ-policy iteration algorithm training over 500 games.

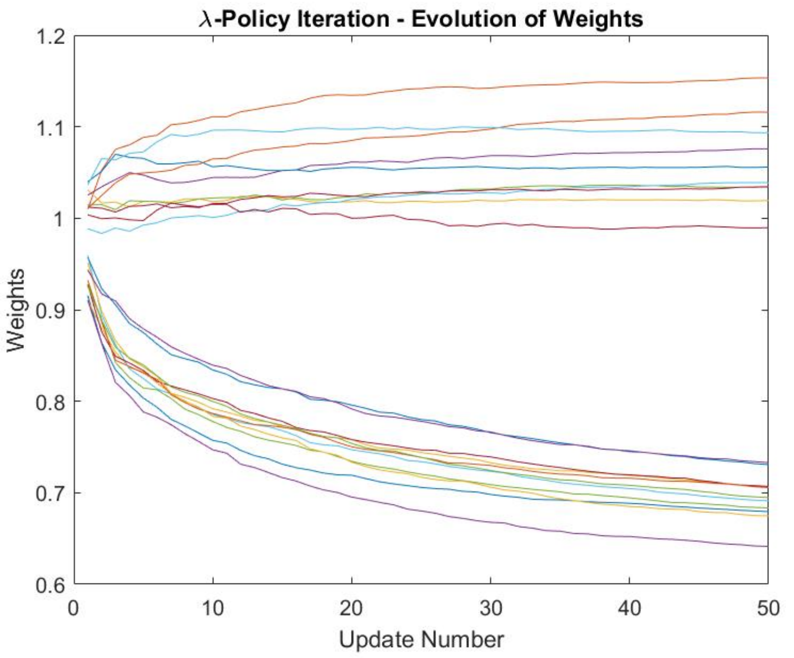

Figure 15: Evolution of the feature vector weights using a λ-policy iteration algorithm training over 500 games.

Finally, the learned weights found with this approach were used to determine control actions for 100 simulated games, with the following results:

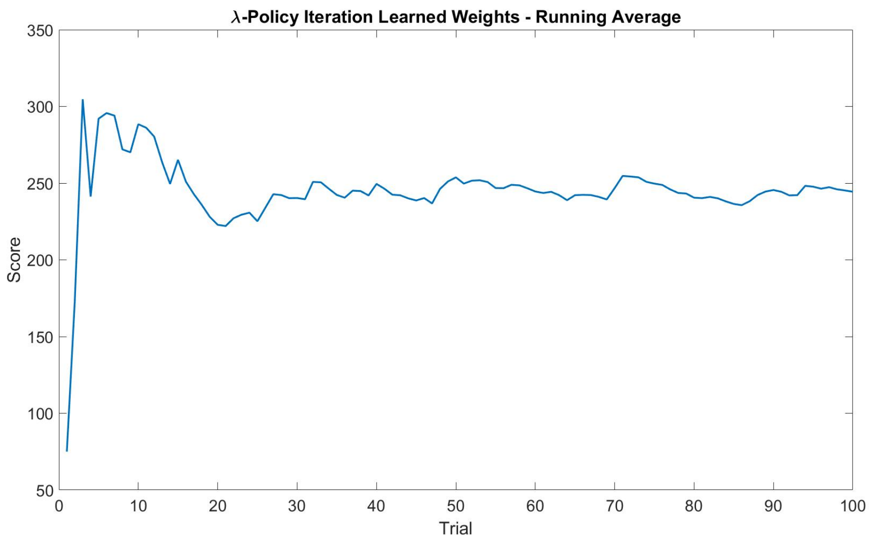

Figure 16: Running average of the number of rows eliminated per game using weights learned by a λ-policy iteration algorithm trained over 500 games.

Clearly, this performance is quite good, and represents an improvement over what we were able to achieve using TD(0). Furthermore, this performance is likely better than what the author could achieve playing Tetris manually, which is quite an exciting benchmark to have surpassed.

Conclusion

In conclusion, Dynamic Programming is a versatile and powerful technique for solving staged optimization problems, with numerous useful extensions that allow it to be used in a variety of settings, as demonstrated through the preceding Tetris examples. With regard to the cases where we face the curse of dimensionality, all hope is not lost. We are able to make use of the understanding we have built up from the simpler settings, leveraging concepts such as the Principle of Optimality to attack otherwise intractable problems. It is clear, however, that this process is quite tricky and complex, and that one must be both very attentive and very persistent when attempting to make use of Approximate Dynamic Programming to circumvent the curse of dimensionality.

Some of this work was completed in collaboration with Max Emerick and Sean Anderson. Credit is owed to Dmitri Bertsekas, Sergey Ioffe, and Benjamin Van Roy, whose papers were very useful as a guide to our first successful implementation of a reinforcement learning algorithm in Section 5(d). Additionally, a special thanks is owed to Jason Marden for his instruction and guidance throughout a challenging and exciting quarter learning the ins and outs of Dynamic Programming.